



developmentor®

A DEVELOPER SERVICES COMPANY

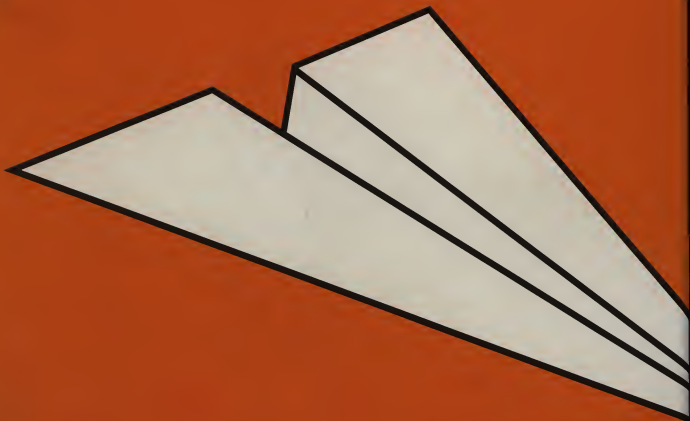
DEVELOP.COM



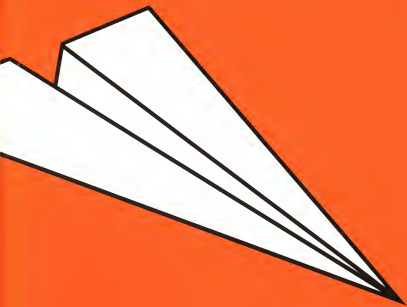
developmentor®

A DEVELOPER SERVICES COMPANY

DEVELOP.COM



Guerrilla .NET
Student Manual
Book 1



developmentor

DEVELOP.COM



Guerrilla .NET

Student Manual

Book 1

Copyright 2001-2002, DevelopMentor, Inc.

For information about other DevelopMentor resources,
please visit: www.develop.com or,

phone us at:

800.669.1932

310.543.1716

08000.562.265

+44.1242.525.108

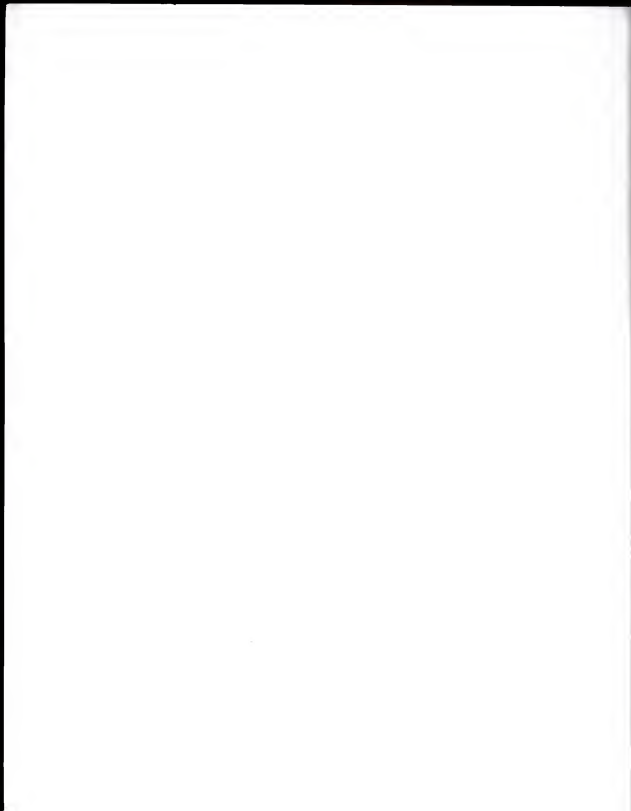
In the US

corporate office direct

within the UK

within Europe

FW100_4.22.02



Guerrilla .NET

Student Manual

Copyright © 2001 DevelopMentor, Inc.

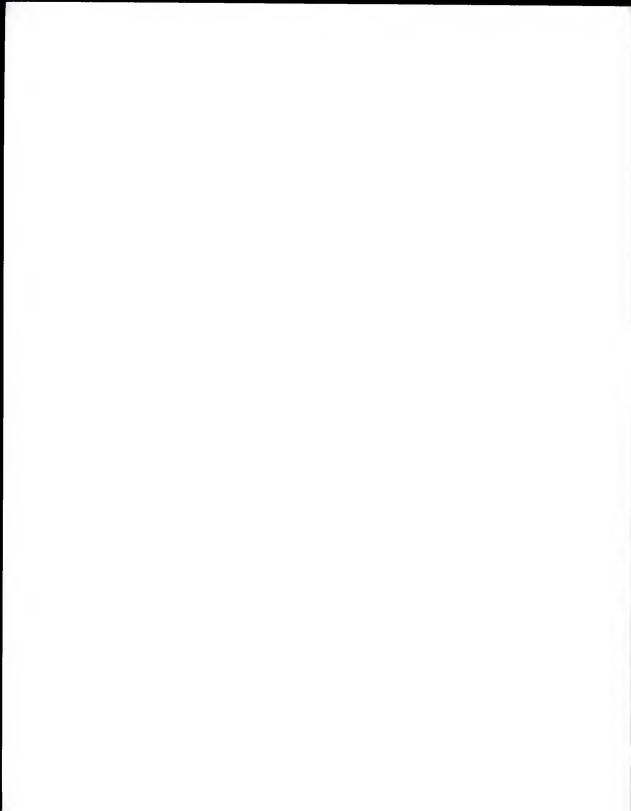
FW100 4-22-02



developmentor™

A DEVELOPER SERVICES COMPANY





Contents

1 .NET Platform Architecture	19
Architecture	21
The evolution of the platform	22
CLR Architecture	28
The role of programming languages	34
The role of type	38
CLS == VB and script	42
Execution scope and the CLR	46
The role of ASP.NET	50
The role of XML, SOAP and Web Services	54
2 Assemblies and Modules	57
Assemblies and the .NET Framework	59
Modules defined	60
Assemblies defined	64
Assembly names	70
Public Keys and Assemblies	74
Loading and resolving assemblies	80
Version policy	84
The assembly cache	90
Assembly resolving via CODEBASE or probing	92
Versioning hazards	98
3 Code Access Security	103
Code Access Security: Motivation	105
History	106
The Component Revolution	110
Authenticode	114
Code Access Security	118
Security Policy	121
Evidence	122
Permissions	124
Policy Levels	128
Code Groups	132
Declarative Permission Requests	136
Administering Policy	142
The CAS Stackwalk	145
The Luring Attack	146
Demand	150
Deny and PermitOnly	152
Assert	156

4	Managed Type Fundamentals	161
	Managed Type Fundamentals.....	163
	The role of type	164
	Access control	168
	Properties	172
	Indexers.....	176
	Interfaces.....	180
	Type Compatibility and Navigation	182
	Explicit Interface Implementation	184
	Base Classes	186
	Base/derived construction	190
	Method Invocation.....	194
5	Reflection and Attributes.....	199
	Reflection	201
	Reflection	202
	System.Type	206
	Using System.Type	210
	Late Binding.....	214
	Activator.CreateInstance.....	216
	FieldInfo	218
	MethodInfo/ConstructorInfo	220
	PropertyInfo.....	224
	Extending type information	226
6	Objects, Values and Memory.....	231
	Objects, Values, and Memory	233
	Reference vs. value types.....	234
	Variables, parameters and fields.....	240
	Cloning.....	246
	Boxing.....	250
	Object lifecycle.....	254
	Finalization.....	258
	Programmer hygiene	262
7	Delegates and Events	267
	Delegates and Events	269
	Motivation	270
	Motivation (cont'd)	274
	Delegates	278
	Declaring Delegates	280
	Using Delegates.....	282
	Supporting Multiple Targets.....	286
	Events.....	290
	Customizing Event Registration	294
	Asynchronous Invocation	296
8	Threads.....	305
	Threads.....	307

Thread Origins	308
Threading Comparisons	314
Thread Synchronization	316
Denial of Service	320
Fairness	322
Read/Write-Aware Synchronization.....	326
Synchronization Primitives	328
Threading and COM Interop.....	332
9 AppDomains and Marshaling.....	337
AppDomains.....	339
Execution scope and the CLR	340
Programming with AppDomains	346
AppDomain events.....	350
AppDomains and the assembly resolver	354
AppDomains and Objects Revisited	360
10 Serialization	365
Serialization	367
Motivation	368
Serialization defined.....	370
When, what, how?.....	372
Client-driven serialization.....	376
Transient data	378
Custom serialization	380
11 Remoting	383
Remoting	385
The Problem Space.....	386
Method remoting overview - client context	388
Method remoting overview - server context.....	390
AppDomain Boundaries and Objects	392
Channels and formatters	396
Channel registration	398
Remoting Details	403
Activation options	404
Working with well known objects.....	406
Configuration files - well known objects.....	410
Client-activated types	416
Configuration files - client-activated types.....	420
Hosting	424
ASP.NET Hosting	426
Metadata Availability	428
Remoting versus Web Services.....	432
12 I/O and Data Transfer	435
Unified I/O	437
I/O Options	438
Polymorphism and I/O	440

	System.Data.IDataReader	442
	System.Data.IDbConnection and friends	446
	System.Xml.XmlReader	450
	System.Xml.XPath.XPathNavigator and friends	456
	System.Data.DataSet	460
13	Server-side programming with ASP.NET	463
	ASP.NET Architecture	465
	Evolution	466
	Revolution	468
	What is ASP.NET?	470
	ASP.NET basics	474
	Page compilation	476
	ASP.NET Compilation model	480
	System.Web.UI.Page	482
	Code behind	486
	ASP.NET directives	490
14	The HTTP Pipeline	495
	HTTP Pipeline	497
	ASP.NET and IIS	498
	HttpHandlers	502
	HttpContext	506
	HttpModules	510
	Implementing an HTTP module	514
	Deploying an HTTP module	516
	Terminating request handling	518
15	WebForms	521
	Server-Side Controls	523
	Traditional HTML generation	524
	ASP.NET server-side controls	528
	Server-side events	532
	Controls	535
	HtmlControls	536
	WebControls	540
	IE Web Controls	544
	Custom Controls	547
	Controls and Pages	548
	Building Custom Controls	552
	Using Custom Controls	554
	Composite Controls	556
	User Controls	560
	Using User Controls	562
16	ASP.NET Security	565
	Introduction to ASP.NET Security	567
	Goals	568
	Security Contexts	570

The Worker Process	572
Client Authentication	576
Principals	578
Authorization	580
Forms Authentication	583
Goals	584
Cookies	586
Basics	588
Mechanics	592
Storing Passwords	596
Postprocessing	598
Protecting Cookies	600
17 ASP.NET Web Services.....	603
ASP.NET Web Services	605
What is a Web Service?	606
What is SOAP?	610
What is XML Schema?	612
The Web Service Description Language (WSDL)	616
Building Web Services.....	618
Web Services in the Raw	620
System.Web.Services	622
Processing Soap Headers	626
SOAP Extensions	630
When WebMethods Fall Short.....	632
Web Service Discovery	636
18 XML Serialization	639
XML Serialization and Schemas	641
XML vs. CLR type system	642
XmlSerializer architecture	646
Type mapping	650
Schema customization	656
Arrays	662
Customizing XmlSerializer.....	666
System.Xml.Schemas.....	670
19 Win32/COM Interoperation	673
Interop Basics	675
Motivation	676
P/Invoke	678
Typed transitions	682
RCW/CCW	688
TLBIMP/TLBEXP	690
REGASM	694
COM+ 1.0	696
Strategies.....	700
20 Appendix: Type Potpourri	705

Arrays	707
Arrays defined	708
Array elements	710
Array capacity	712
System.Array	716
IComparable	720
Text Basics	723
System.String	724
ToString	728
System.Text.StringBuilder	730
String.Format	734
System.IFormattable	738
Collections	743
IEnumerable/IEnumerator	744
Programming languages and IEnumerable	748
ICollection and friends	752
ArrayList and friends	756
Glossary	761
Bibliography	765

Figures

Figure 1.1: The evolution of the Microsoft platform	23
Figure 1.2: Cross-host vs. in-memory integration.....	26
Figure 1.3: Control vs. productivity	29
Figure 1.4: The evolution of the loader.....	30
Figure 1.5: CLR Architecture	31
Figure 1.6: Base class library	32
Figure 1.7: You and the Common Language Runtime	33
Figure 1.8: .NET language features.....	36
Figure 1.9: Hello, World - C#	36
Figure 1.10: Hello, World - VB.NET	37
Figure 1.11: Building and using CLR components	37
Figure 1.12: Sample MAKEFILE.....	37
Figure 1.13: Type fragmentation under COM.....	39
Figure 1.14: Pervasive Type in the CLR	39
Figure 1.15: The CLR type system	40
Figure 1.16: Using the type system	41
Figure 1.17: Type subsetting with the Common Language Specification	43
Figure 1.18: CLS compliance.....	43
Figure 1.19: Objects, AppDomains, and Processes	47
Figure 1.20: Hello, World - C#/ASP.NET	51
Figure 1.21: HelloWorld.ASHX - Hello, World in C#/ASP.NET (revisited)	52
Figure 1.22: Building and using ASP.NET components.....	52
Figure 1.23: Type segregation under XML.....	55
Figure 2.1: CLR Module Format.....	61
Figure 2.2: Modules and Assemblies	66
Figure 2.3: Multi-module assemblies using CSC.EXE	66
Figure 2.4: Multi-module assemblies using CSC.EXE and NMAKE.....	67
Figure 2.5: A multi-module assembly.....	68
Figure 2.6: Inside the AssemblyVersion attribute	72
Figure 2.7: Fully specified assembly names	73
Figure 2.8: Managing public/private keys using SN.EXE.....	76
Figure 2.9: Strong assembly references	77
Figure 2.10: Delay signing an assembly	78
Figure 2.11: Loading an assembly with an explicit CODEBASE	82
Figure 2.12: Assembly resolution and loading	83
Figure 2.13: Loading an assembly using the assembly resolver	83
Figure 2.14: Assembly resolver configuration file format	86
Figure 2.15: Setting the version policy	87
Figure 2.16: Version Policy.....	88

Figure 2.17: Setting the application to safe-mode	88
Figure 2.18: Global Assembly Cache	91
Figure 2.19: Specifying the codebase using configuration files	93
Figure 2.20: APPBASE and the relative search path	94
Figure 2.21: Setting the relative search path	94
Figure 2.22: Culture-neutral probing	95
Figure 2.23: Culture-dependent probing	96
Figure 2.24: Assembly resolution	97
Figure 3.1: Monolithic applications	107
Figure 3.2: Component-based applications	111
Figure 3.3: Well-meaning, but ultimately bad code	115
Figure 3.4: CAS is layered on top of OS native security	119
Figure 3.5: Some built-in classes of evidence	123
Figure 3.6: Some built-in code access security permission classes	125
Figure 3.7: Example: requesting a permission	126
Figure 3.8: Policy levels	130
Figure 3.9: A policy level consists of a tree of code groups	133
Figure 3.10: Modeling a boolean AND with two code groups	134
Figure 3.11: Modeling a boolean OR with two code groups	135
Figure 3.12: Annotating an assembly with a permission refusal	138
Figure 3.13: Annotating an assembly with permission requests	139
Figure 3.14: Creating a serialized permission set	139
Figure 3.15: MSCORCFG.MSC MMC Snap-In	143
Figure 3.16: The IStackWalk interface	148
Figure 3.17: Example: denying permissions before making a call	153
Figure 3.18: Deny versus Assert in a stackwalk	158
Figure 3.19: How to remove CAS stack frame modifiers	158
Figure 4.1: Type definitions in C#	165
Figure 4.2: Distinguishing types via flags and base types	166
Figure 4.3: Access modifiers	170
Figure 4.4: Access modifiers	170
Figure 4.5: Declaring a property	174
Figure 4.6: Using a property	174
Figure 4.7: Declaring an indexer	177
Figure 4.8: Using an indexer	178
Figure 4.9: Declaring and implementing interfaces	181
Figure 4.10: Using interfaces	181
Figure 4.11: Type navigation operators	183
Figure 4.12: Explicit interface implementation	185
Figure 4.13: Specifying a base class in C#	187
Figure 4.14: base types and constructors	191
Figure 4.15: Derivation and construction	192
Figure 4.16: Perils of base construction	193
Figure 4.17: C# method modifiers	195
Figure 4.18: Using method modifiers	196
Figure 4.19: Combining method modifiers	197

Figure 4.20: Combining method modifiers	197
Figure 5.1: The role of reflection	204
Figure 5.2: Using reflection	204
Figure 5.3: Pervasive type and GetType	207
Figure 5.4: Pervasive type and System.Type	208
Figure 5.5: Reflection and the CLR type system	211
Figure 5.6: Reflection object model	212
Figure 5.7: Walking every element in an assembly	212
Figure 5.8: Using reflection to instantiate a type	217
Figure 5.9: Passing constructor arguments to CreateInstance	217
Figure 5.10: System.Reflection.FieldInfo	219
Figure 5.11: Using System.Reflection.FieldInfo	219
Figure 5.12: Using System.Reflection.MethodInfo	221
Figure 5.13: Using System.Reflection.MethodInfo	222
Figure 5.14: Using System.Reflection.PropertyInfo	225
Figure 5.15: Specifying an attribute	227
Figure 5.16: Defining a custom attribute	228
Figure 5.17: Retrieving attributes	229
Figure 6.1: The CLR type system	236
Figure 6.2: C# and VB.NET built-in types	237
Figure 6.3: Defining a value type as a structure	238
Figure 6.4: Defining a value type as an enumeration	238
Figure 6.5: Defining a value type as a bitfield	239
Figure 6.6: Using value and reference types	242
Figure 6.7: Reference and value types	242
Figure 6.8: Using reference types	243
Figure 6.9: Reference types and assignment	243
Figure 6.10: Using value types	244
Figure 6.11: Values and assignment	244
Figure 6.12: Pass by value parameters	245
Figure 6.13: Pass by reference parameters	245
Figure 6.14: System.ICloneable	247
Figure 6.15: Shallow copy	247
Figure 6.16: Implementing System.ICloneable	248
Figure 6.17: Deep copy	248
Figure 6.18: Implementing System.ICloneable	249
Figure 6.19: Implementing System.ICloneable using new	249
Figure 6.20: Boxing	251
Figure 6.21: Boxing and unboxing	252
Figure 6.22: Root and non-root references	256
Figure 6.23: Liveness and garbage collection	257
Figure 6.24: Implementing System.Object.Finalize in C#	260
Figure 6.25: System.IDisposable	263
Figure 6.26: Implementing Dispose	263
Figure 6.27: References and deterministic finalization	264
Figure 6.28: The C# using statement	264

Figure 6.29: C#'s using statement	265
Figure 7.1: A class-based callback relationship: caller	271
Figure 7.2: A class-based callback relationship: target	271
Figure 7.3: A class-based callback relationship: registration	272
Figure 7.4: An interface used for callbacks	275
Figure 7.5: An interface-based callback relationship: caller	275
Figure 7.6: An interface-based callback relationship: target	275
Figure 7.7: Delegate type hierarchy	279
Figure 7.8: Declaring user-defined delegate	281
Figure 7.9: C# to CLR language mapping	281
Figure 7.10: Calling through a delegate	283
Figure 7.11: Simplified target	283
Figure 7.12: Delegate initialization	284
Figure 7.13: A delegate instance	284
Figure 7.14: Multiple target registration	287
Figure 7.15: Iterating through registered targets	288
Figure 7.16: A multicast delegate	289
Figure 7.17: Declaring events	291
Figure 7.18: Event registration	292
Figure 7.19: Customizing event [un]registration	295
Figure 7.20: C# to CLR language mapping - asynchronous invocation	297
Figure 7.21: Dealing with lengthy operations	298
Figure 7.22: Async invocation - fire-and-forget	298
Figure 7.23: Async delegates - fire-and-forget	299
Figure 7.24: Async invocation - polling for completion	300
Figure 7.25: Async delegates - polling	300
Figure 7.26: Async invocation - blocking	301
Figure 7.27: Async invocation - completion callback	302
Figure 7.28: Async delegates - callback	302
Figure 8.1: Threading using delegates	309
Figure 8.2: Threading using the ThreadPool class	310
Figure 8.3: Threading using timers	311
Figure 8.4: Threading using the Thread class	312
Figure 8.5: Comparing asynchronous execution alternatives	315
Figure 8.6: Using a Monitor	317
Figure 8.7: Protecting Statics	318
Figure 8.8: Defending against external SyncBlock acquisition	321
Figure 8.9: Using a Monitor to Prevent Starvation	324
Figure 8.10: Using ReaderWriterLock	327
Figure 8.11: Using the Interlocked class	329
Figure 8.12: Using WaitHandle.WaitAll and mutexes	331
Figure 8.13: ApartmentState enumeration	334
Figure 8.14: Selecting an apartment type for the main thread	334
Figure 8.15: Selecting an apartment type for the main thread	335
Figure 9.1: Objects, AppDomains, and Processes	341
Figure 9.2: Scoping types and statics to AppDomains	342

Figure 9.3: AppDomains and threads	344
Figure 9.4: System.AppDomain (excerpt)	347
Figure 9.5: Spawning new applications.....	348
Figure 9.6: Calling into foreign AppDomains	349
Figure 9.7: AppDomain events.....	351
Figure 9.8: Retro-programming in C#.....	352
Figure 9.9: AppDomain environment properties	355
Figure 9.10: A simple application (child.exe)	358
Figure 9.11: Implementing shadow copy (host.exe).....	358
Figure 9.12: Loading using shadow copy	359
Figure 9.13: Agility and objects	361
Figure 9.14: Marshaling objects across AppDomains.....	362
Figure 9.15: Cross-AppDomain marshaling	363
Figure 10.1: Case study.....	373
Figure 10.2: Client-driven serialization	377
Figure 10.3: Client-driven deserialization	377
Figure 10.4: [NonSerialized] and IDeserializationCallback	379
Figure 10.5: Implementing ISerializable	381
Figure 11.1: Method Remoting Overview.....	391
Figure 11.2: Agility and objects	393
Figure 11.3: Marshaling objects across AppDomains.....	394
Figure 11.4: Channel Registration.....	399
Figure 11.5: Exposing an object via remoting	399
Figure 11.6: Cross-domain method calls	400
Figure 11.7: Dispatching cross-host method calls.....	401
Figure 11.8: Accessing a remote object.....	402
Figure 11.9: The ICalc interface.....	407
Figure 11.10: Well Known Object - Server	408
Figure 11.11: Well Known Object - Client	409
Figure 11.12: Remoting configuration file - client.....	411
Figure 11.13: Revised client - using configuration file	412
Figure 11.14: Remoting configuration file - server	413
Figure 11.15: Revised server - using configuration file	413
Figure 11.16: Supporting client-activated types - server	417
Figure 11.17: Using a client-activated type without operator new	418
Figure 11.18: CAO client using operator new.....	419
Figure 11.19: Remoting configuration file - client.....	421
Figure 11.20: Remoting configuration file - server	422
Figure 11.21: web.config for use with ASP.NET Hosting	427
Figure 11.22: WSDL-enabled configuration file	431
Figure 11.23: SOAPSUDS command line	431
Figure 11.24: .NET Remoting versus Web Services	433
Figure 12.1: Streaming I/O types	439
Figure 12.2: Polymorphic I/O	441
Figure 12.3: IDataRecord/IDataReader	443
Figure 12.4: IDataReader/IDataRecord	444

Figure 12.5: Using IDataRecord/IDataReader.....	444
Figure 12.6: IDbConnection and friends.....	447
Figure 12.7: Using IDbConnection and friends.....	448
Figure 12.8: Using IDbCommand with parameters.....	448
Figure 12.9: Streaming I/O Model.....	452
Figure 12.10: XmlReader.....	453
Figure 12.11: Using XmlReader.....	453
Figure 12.12: Using XmlReader.....	454
Figure 12.13: Opening an XmlReader.....	454
Figure 12.14: Using XmlWriter.....	455
Figure 12.15: Opening an XmlWriter.....	455
Figure 12.16: XmlNode Hierarchy.....	457
Figure 12.17: XmlDocument.Load.....	458
Figure 12.18: Loading a DOM.....	458
Figure 12.19: Using a DOM.....	458
Figure 12.20: XPathNavigator.....	459
Figure 12.21: Using XPathNavigator and XPath.....	459
Figure 13.1: High-level view of ASP.NET.....	472
Figure 13.2: ASP.NET page.....	475
Figure 13.3: ASP.NET Page Compilation.....	478
Figure 13.4: System.Web.UI.Page.....	483
Figure 13.5: System.Web.UI.Page.....	484
Figure 13.6: Sample aspx file customizing Page.....	484
Figure 13.7: Sample aspx file with code behind.....	487
Figure 13.8: Sample code-behind file - SamplePage.cs.....	488
Figure 13.9: ASP.NET Directives.....	491
Figure 13.10: @page Directives.....	492
Figure 14.1: ASP.NET Architecture.....	500
Figure 14.2: Classes in the HTTP Pipeline of ASP.NET.....	501
Figure 14.3: Web.config file defining custom HttpHandler class.....	503
Figure 14.4: IHttpHandler/IHttpHandlerFactory.....	504
Figure 14.5: Sample IHttpHandler implementation.....	504
Figure 14.6: HttpContext Properties.....	507
Figure 14.7: HttpContext Example.....	507
Figure 14.8: Web Application Object Model.....	511
Figure 14.9: The IHttpModule interface.....	511
Figure 14.10: System-provided modules.....	512
Figure 14.11: Implementing an HTTP module.....	515
Figure 14.12: Configuring HttpModules.....	517
Figure 14.13: Calling CompleteRequest.....	519
Figure 15.1: Traditional ASP page.....	526
Figure 15.2: ASP.NET page using server-side controls.....	529
Figure 15.3: HTML generated by server-side controls.....	530
Figure 15.4: Using server-side events.....	533
Figure 15.5: HTML generated using server-side events.....	534
Figure 15.6: Hierarchy of HtmlControls and the tags they map to.....	537

Figure 15.7: A sample ASP.NET page written with HtmlControls	538
Figure 15.8: Hierarchy of WebControls.....	541
Figure 15.9: WebControl catalog.....	542
Figure 15.10: A sample ASP.NET page written with WebControls	543
Figure 15.11: IE WebControl catalog	545
Figure 15.12: IE WebControls hierarchy.....	546
Figure 15.13: A sample ASP.NET page written with IE WebControls	546
Figure 15.14: An .ASPX page prior to compilation	549
Figure 15.15: In-memory representation of an .ASPX page.....	550
Figure 15.16: Important members of the System.Web.UI.Control class	551
Figure 15.17: A simple custom control	553
Figure 15.18: Client .aspx page for the SimpleControl custom server control	555
Figure 15.19: Composite Control example	558
Figure 15.20: A User Control.....	561
Figure 15.21: A sample client to a user control.....	563
Figure 16.1: Security Contexts	571
Figure 16.2: The Worker Process.....	573
Figure 16.3: Configuring the worker process security context	573
Figure 16.4: Propagating the thread security context	574
Figure 16.5: IPPrincipal and IIdentity	579
Figure 16.6: Url-based authorization in web.config	581
Figure 16.7: Declarative principal demands	581
Figure 16.8: Imperative principal demands.....	582
Figure 16.9: Enabling Basic Forms Authentication.....	589
Figure 16.10: A Simple Login Form.....	589
Figure 16.11: Handling the Login.....	590
Figure 16.12: Transparent conversion of cookies to contexts.....	593
Figure 16.13: Transparent redirection to login page	594
Figure 16.14: Storing hashed passwords in web.config	597
Figure 16.15: Creating hashed passwords for the config file	597
Figure 16.16: Looking up Passwords	597
Figure 16.17: Adding Roles using global.asax.....	599
Figure 17.1: Web Service Protocols	607
Figure 17.2: The Web Service Promise	608
Figure 17.3: SOAP Request/Response	611
Figure 17.4: A SOAP Envelope	611
Figure 17.5: XML Without Schema	613
Figure 17.6: XML With Schema	614
Figure 17.7: WSDL.....	617
Figure 17.8: .NET Web Services Architecture	623
Figure 17.9: Implementing a WebMethod in a .ASMX File	624
Figure 17.10: Using WSDL.EXE to Generate a Proxy.....	625
Figure 17.11: SOAP Envelope Headers.....	627
Figure 17.12: Defining SOAP Headers.....	627
Figure 17.13: Processing SOAP Headers	628
Figure 17.14: Setting SOAP Headers on Client	628

Figure 17.15: SOAP Extension Architecture.....	631
Figure 17.16: Custom SOAP Handlers.....	633
Figure 17.17: A Custom SOAP Handler.....	634
Figure 17.18: UDDI in Action.....	637
Figure 18.1: Bridging the XML type system.....	643
Figure 18.2: Schema-centric type mapping.....	644
Figure 18.3: Class-centric type mapping.....	644
Figure 18.4: XmlSerializer object model.....	648
Figure 18.5: XmlSerializer (excerpt).....	648
Figure 18.6: XmlSerializer.Serialize example.....	649
Figure 18.7: XmlSerializer.Deserialize example.....	649
Figure 18.8: XmlRootAttribute mapping.....	652
Figure 18.9: [XmlRoot]/[XmlType] parameters.....	653
Figure 18.10: [XmlRoot]/[XmlType] example.....	653
Figure 18.11: [XmlRoot]/[XmlType] example output.....	654
Figure 18.12: XSD.EXE.....	654
Figure 18.13: XmlAttributeAttribute/XmlElementAttribute parameters.....	657
Figure 18.14: XmlElementAttribute mapping.....	658
Figure 18.15: XmlAttribute mapping.....	659
Figure 18.16: [XmlAttribute]/[XmlElement] example.....	660
Figure 18.17: [XmlAttribute]/[XmlElement] example output.....	660
Figure 18.18: [XmlEnum] example.....	661
Figure 18.19: [XmlEnum] example output.....	661
Figure 18.20: [XmlArrayItem] example.....	663
Figure 18.21: [XmlArrayItem] example output.....	664
Figure 18.22: UnknownAttribute handler.....	667
Figure 19.1: Integrating managed and unmanaged code.....	677
Figure 19.2: DllImport attribute parameters.....	679
Figure 19.3: Using DllImport.....	680
Figure 19.4: Using DllImport with PreserveSig.....	681
Figure 19.5: Crossing the boundary.....	683
Figure 19.6: Isomorphic and non-isomorphic types.....	684
Figure 19.7: Crossing the boundary with non-isomorphic parameters.....	685
Figure 19.8: MarshalAs attribute parameters.....	686
Figure 19.9: MarshalAs with parameters.....	686
Figure 19.10: MarshalAs.....	687
Figure 19.11: MarshalAs equivalent in IDL.....	687
Figure 19.12: RCW/CCW architecture.....	689
Figure 19.13: TLBIMP/TLBEXP.....	691
Figure 19.14: C# as a better IDL.....	692
Figure 19.15: C# as a better IDL (generated TLB).....	693
Figure 19.16: Hosting the CLR from VB.....	695
Figure 19.17: System.EnterpriseServices.ServicedComponent.....	697
Figure 19.18: System.EnterpriseServices.ContextUtil.....	698
Figure 19.19: A COM+ 1.0 configured class.....	699
Figure 19.20: .NET via partial source-code porting.....	701

Figure 19.21: .NET via full source-code porting.....	702
Figure 19.22: .NET via binary import (punt approach).....	703
Figure 20.1: Creating and using an array.....	709
Figure 20.2: Array initialization	709
Figure 20.3: Single-dimensional array of value type	711
Figure 20.4: Single-dimensional array of reference type	711
Figure 20.5: Rectangular multi-dimensional array.....	713
Figure 20.6: Creating and using a multi-dimensional array.....	714
Figure 20.7: Multi-dimensional array initialization	714
Figure 20.8: Jagged multi-dimensional array.....	715
Figure 20.9: Creating and using a jagged array	715
Figure 20.10: System.Array (excerpt)	717
Figure 20.11: Arrays and type-compatibility	718
Figure 20.12: System.Array (excerpt)	719
Figure 20.13: Using System.Array	719
Figure 20.14: System.IComparable.....	721
Figure 20.15: Implementing System.IComparable	721
Figure 20.16: System.Array (excerpt)	722
Figure 20.17: Using System.Array revisited	722
Figure 20.18: System.String (excerpt).....	725
Figure 20.19: Simple string manipulation	726
Figure 20.20: String comparison.....	726
Figure 20.21: System.String miscellany	727
Figure 20.22: Implementing ToString.....	729
Figure 20.23: System.Text.StringBuilder (excerpt).....	731
Figure 20.24: System.Text.StringBuilder.....	732
Figure 20.25: Simple string manipulation using System.Text.StringBuilder...	732
Figure 20.26: System.String.Format	735
Figure 20.27: Format string placeholder syntax	735
Figure 20.28: String.Format numeric codes	739
Figure 20.29: String.Format picture codes.....	740
Figure 20.30: System.IFormattable	741
Figure 20.31: Implementing System.IFormattable	741
Figure 20.32: Inside Console.WriteLine	741
Figure 20.33: IEnumerable/IEnumerator	745
Figure 20.34: Using IEnumerable/IEnumerator.....	745
Figure 20.35: IEnumerable/IEnumerator	746
Figure 20.36: foreach	749
Figure 20.37: A pseudo-implementation of IEnumerable	750
Figure 20.38: A plain vanilla implementation of IEnumerable	751
Figure 20.39: Collection interfaces	753
Figure 20.40: ICollection	754
Figure 20.41: IList	754
Figure 20.42: Using IList.....	754
Figure 20.43: IDictionary	755
Figure 20.44: Using IDictionary.....	755

Figure 20.45: Built-in collection types	757
Figure 20.46: Built-in collection types (revisited)	758

Module 1

.NET Platform Architecture

Microsoft .NET is a new platform for integrating software components from multiple organizations across heterogeneous systems. The .NET platform consists of two technologies: The Common Language Runtime (CLR) and XML-based Web Services. The CLR provides a pervasive type system that spans programming language and operating system boundaries and is used for in-memory integration of components. Web Services are server-based applications that expose their functionality using XML, HTTP and SOAP.

After completing this module, you should be able to:

- ❑ understand where the Common Language Runtime came from
- ❑ understand what problems the Common Language Runtime solves
- ❑ understand the role of type in CLR programming
- ❑ code and build simple CLR programs

Architecture

The .NET Framework/Common Language Runtime (CLR) provide a new technology for integrating software components from multiple organizations. The CLR provides a pervasive type system that spans programming language and operating system boundaries.

The evolution of the platform

.NET is the next evolutionary step for the Microsoft platform

- Biggest platform shift since move from DOS to Windows NT
- Departure from Win32, classic C++ and MSVBVM60.DLL
- Departure from COM and DCOM
- Internet standards used from cross-machine integration
- Common Language Runtime (CLR) used from same-machine integration
- CLR raises programming model to new level of abstraction

Every year, major platform vendors launch broad "marketectural" visions to recast their existing technologies as something fundamentally new. To this end, it is easy to dismiss Microsoft's .NET initiative as yet another content-free marketing ploy engineered to fool the trade press, Wall Street analysts, and gullible CIOs. However, every so often, a vendor opts to completely change the platform, hopefully for the better. The last such platform change from Microsoft was the shift from DOS to Windows NT. The .NET initiative represents a platform shift that is equally broad and deep.

The .NET initiative represents a departure from the Win32/COM platform that dominated the Microsoft landscape during the 1990's. As shown in figure 1.1, the technologies that were used to build software components during that era included Win32, C++, MSVBVM60.DLL, COM and DCOM. Each of these technologies has reached a plateau that they are unlikely to transcend. The reasons are as follows:

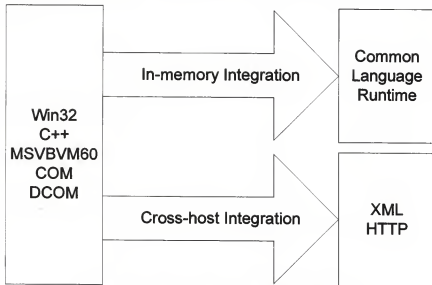


Figure 1.1: The evolution of the Microsoft platform

Win32: Win32 was designed around the Windows NT architecture. If your code only needs to run on Windows NT/2000, then this is not a huge problem. If your code needs to run on other platforms or devices not capable of supporting Windows NT/2000, then your Win32-based code may be stranded. Adapting the Win32 API to architectures with different concurrency, I/O and memory management models has proven to be less than appealing, as is evidenced by the relative lack of success of Win32 implementations on Unix. Additionally, the Win32 API has a hard dependency on a 32-bit address space. With the

advent of Intel's IA-64 processor and Windows.NET server, this assumption no longer holds true for all variations of Windows NT.

C++: Actually, the C++ compiler was the problem here. The C++ compiler had no respect for your programs. Most modern C++ compilers take the well-designed abstractions and designs that are present in your source code and completely obfuscate them into machine code that bears little resemblance to the programmer's work. If all that is needed is raw execution, then the current state of affairs is acceptable. However, as developers come to depend on the underlying platform to provide more services to their code at runtime, the platform needs to better understand programmer intentions. This implies that the programmer's original abstraction needs to be retained in binary form for consumption by the supporting platform. This binary form needs to include things like type relationships, method signatures, and any service-specific annotations that may be required.

MSVBVM60.DLL: This DLL was the support library that all programs written using Visual Basic 6.0 relied upon. This DLL had evolved considerably over the years and was adapted to run as well as possible in 32-bit, multithreaded, server-side scenarios. Unfortunately, the adaptation was less than ideal. In particular, aspects of MSVBVM60.DLL made it difficult to deploy VB6-based code in several environments due to the high degree of thread-affinity that permeates MSVBVM60.DLL.

COM: The Component Object Model began its life in 1988. In 1988, the DOS-based Microsoft platform was barely functional, especially when you consider the state of the hardware platform during that era. In order to adapt to the then-state of the platform, COM began life as a minimal technology that simply tried to add type-awareness to the loader. `CoCreateInstance` replaced `LoadLibrary`. `QueryInterface` replaced `GetProcAddress`. COM's main weakness was its lack of a unified metadata format, in which a single representation of a type description could be considered definitive or normative. During much of COM's life, there were two competing formats: text-based IDL files and binary TLB files. Unfortunately, neither was a proper superset of the other, which meant that in some situations, you needed an IDL file, and in others, a TLB was necessary. As the COM platform tried to provide more services over time (via Microsoft Transaction Server (MTS) and COM+), the lack of a normative, extensible metadata format became a major obstacle to innovation. The poor state of COM metadata was the primary motivation for the Component Object Runtime (COR), which eventually was renamed to the Common Language Runtime (CLR) and is the heart of the .NET platform.

DCOM: The Distributed Component Object Model took the idea of interface-based programming made popular by COM and asserted that component interfaces could be transparently adapted to act as network protocols. While this idea is technically feasible, the implementation of DCOM turned out to have more than its fair share of technical issues. For one, the specification of DCOM was a minor extension to the platform-neutral Open Software

Foundation's Distributed Computing Environment's Remote Procedure Call (OSF DCE RPC) protocol. However, the reality of Microsoft's implementation of DCOM required an NT-based authentication service to be present, which rendered DCOM a largely NT-only solution. Additionally, the implementation of DCOM security tended to be very deployment sensitive, which made it hard to adapt to all but the most well-administered network environments. Finally, application protocols such as DCOM or CORBA's IIOP tend to be thwarted by the firewall infrastructure present in most large corporations, rendering these protocols useless for applications that need to achieve Internet-scale reach.

As just described, each of these technologies has run into their own technological dead-ends. The .NET initiative is Microsoft's way of "cleaning house" and building a new platform for component development and integration.

The .NET initiative is a bifurcated platform, in which in-memory integration is radically different from cross-process/machine integration. For cross-process/cross-machine integration, .NET departs from DCOM in favor of Internet-friendly protocols such as HTTP, XML, and SOAP. For in-memory component integration, .NET departs from Win32 and COM in favor of the Common Language Runtime (CLR).

Microsoft's move from COM and DCOM to the CLR and SOAP is a dramatic change of approach. In the 1990's, the in-memory component technology (COM) required virtually no runtime support and could be easily adapted to a variety of runtime environments (e.g., Netscape's Navigator/Mozilla contain XPCOM, a portable open-source implementation of in-memory COM). In the .NET era, the in-memory component technology (CLR) requires a new compiler, a new executable file format, and has its own unique runtime semantics. This is a reasonable step for Microsoft to take, since Microsoft can make broad, sweeping changes to the Windows platform, given that they control the operating system, server-side application containers, the web browser, and the development tools used by all but a handful of Windows developers. By way of contrast, Microsoft is actually loosening its grip on how components are integrated across host boundaries.

In the 1990's, Microsoft's cross-machine integration technology (DCOM) required a significant amount of NT-specific runtime support in order to actually work, which made hosting DCOM in non-Windows NT environments impractical, especially across organizational boundaries or over the Internet. In the .NET era, Microsoft's preferred cross-machine integration technology (SOAP) uses off-the-shelf technologies (HTTP and XML) that are freely available for virtually any platform or environment in use today. This change of overall strategy is illustrated in figure 1.2.

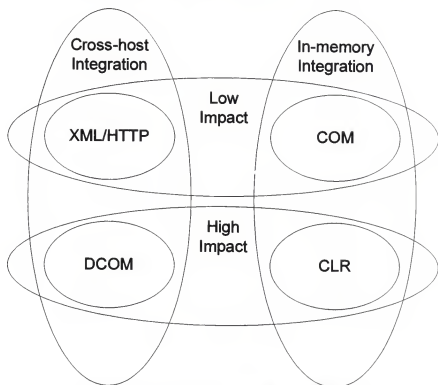


Figure 1.2: Cross-host vs. in-memory integration



CLR Architecture

The CLR was designed to replace the Win32 and COM loader and component metadata

- "Managed" components (a.k.a. assemblies) have rich, extensible type/dependency information
- Managed components rely on MSCOREE.DLL (the runtime)
- Managed components rely on MSCORLIB.DLL (the runtime library)
- MSCOREE.DLL is a "native" DLL that loads managed DLLs
- MSCORLIB.DLL is a "managed" DLL that contains the core types used throughout the system
- MSCOREE.DLL is actually a shim to MSCORWKS.DLL or MSCORSRV.DLL

mscorlib.dll - execution enviro, written in c++

The platform shift from Win32/COM to the CLR is as significant as the move from the "real-mode" environment of DOS to the "protected-mode" environment of Windows NT. There is a place for developers who choose to work in terms of physical memory and interrupt handlers, which is why there is a kernel-mode device driver infrastructure in Windows NT. However, few developers need the level of control offered by kernel-mode, nor can many developers write code that can function properly in kernel-mode without causing the infamous "blue-screen-of-death" caused by kernel-mode code gone awry. Rather, most developers take for granted the conveniences of virtual memory management and thread-based concurrency afforded to user-mode programs.

What is ironic is that writing user-mode code productively and effectively is only marginally easier than writing kernel-mode code. In both cases, the developer is distracted by a morass of tedious details that have nothing to do with the problem domain of the application. This is why environments such as Java and Visual Basic have relegated C/C++ coding to a shrinking number of low-level systems programmers.

The CLR formalizes the distinction between low-level systems programmers and high-level application programmers by introducing a new programming model and platform that eschews memory and threads in favor of types, objects, and values. This distinction is illustrated in figure 1.3. Programmers who move to the CLR are encouraged to view the world in terms of types, objects, and values. This does not mean that virtual memory or threads no longer exist. However, it does mean that programmers should stop dealing with them explicitly.

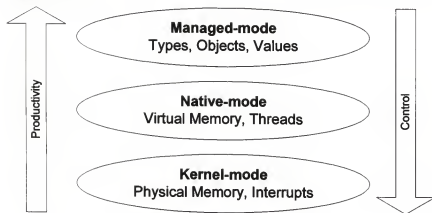


Figure 1.3: Control vs. productivity

The CLR replaces the Win32 and COM loaders with a new type-oriented loader that relies on high-fidelity, extensible component metadata. To write components for the CLR, a new compiler and linker is needed that understands

this new metadata format. Additionally, the CLR takes over execution of component code in order to provide richer services than are possible with raw machine code execution.

Code written for the CLR executes in managed execution mode. Under managed execution, the CLR is able to track calls from one component to another, enforcing security, concurrency and resource management policies. While it is possible for CLR-based programs to drop down to unmanaged execution mode, it is preferable to avoid the managed/unmanaged transition for performance reasons, just as user/kernel transitions are to be avoided in normal systems programming.

In many ways, the CLR can be seen as a natural evolution of COM. The fundamental characteristic of COM was its layering of type over the underlying OS loader (e.g., LoadLibrary/GetProcAddress). As shown in figure 1.4, the CLR extends that idea by further integrating type information into the underlying executable file format. Note that in each of these formats, the executable file format begins with an MZ header, which allows DOS-based loaders to recognize the file as a valid executable. At each point of evolution, Microsoft took pains to keep some degree of backwards-compatibility, even though the MZ stub simply prints out the message "This program cannot be run in DOS mode." when loaded by a pre-Windows version of DOS.

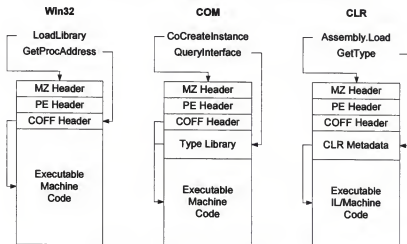


Figure 1.4: The evolution of the loader

All CLR-based code must be packaged into an assembly, which is used to encapsulate code into a binary component. A CLR assembly requires the presence of the Common Language Runtime in order to function. The CLR is composed of two primary components: `MSCOREE.DLL` and `MSCORLIB.DLL`. `MSCOREE.DLL` is sometimes called "the shim" as it is a fairly thin veneer over

the actual implementation of the CLR. `MSCOREE.DLL` is a native COM DLL that exports a small number of API functions (e.g., `CorBindToRuntimeEx`) that are used to host the CLR from native code. Upon initialization, `MSCOREE.DLL` simply loads the "real" runtime, which is implemented in `MSCORWKS.DLL` (uniprocessor) or `MSCORSVR.DLL` (multiprocessor). As is shown in figure 1.5, all CLR-based code has a PE/COFF dependency on the shim (`MSCOREE.DLL`). The shim determines which version of the "real" runtime to load based on the underlying hardware as well as configuration options. Because `MSCOREE.DLL` is the primary DLL used to access the CLR, the remainder of this text will use the term `MSCOREE` generically to refer to the shim as well as whichever version of the runtime is actually loaded.

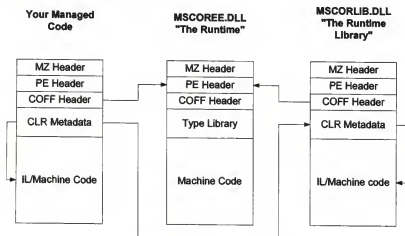


Figure 1.5: CLR Architecture

If the architects of the CLR are successful, `MSCOREE` will be the last Win32/COM DLL ever produced. All new DLLs will use the new metadata format, which places a COFF-level dependency on `MSCOREE.DLL` and friends. If `MSCOREE` is the last Win32/COM DLL produced, then `MSCORLIB.DLL` is the first CLR-based DLL ever produced. `MSCORLIB.DLL` is a managed component that contains the metadata for the common type system as well as a large number of library routines. `MSCORLIB.DLL` exposes types and methods from multiple namespaces, as shown in figure 1.6. Note that the CLR ships with a family of runtime libraries that provide a language-neutral way to write web programs, database access programs, XML programs and Windows programs. All but a few of these libraries reside in `MSCORLIB.DLL`, the core system library that virtually every CLR-based program relies upon.

Namespace	Purpose	Assembly DLL
System	Core type system	mscorlib
System.Reflection	Runtime type info	mscorlib
System.Security	Access control	mscorlib
System.Text	Text encoding/munging	mscorlib
System.Collections	Collection types	mscorlib
System.IO	Binary and text I/O	mscorlib
System.Threading	Threading/locking	mscorlib
System.Runtime.Serialization	Object persistence	mscorlib
System.Runtime.Remoting	SOAP/AOP/Proxies	mscorlib
System.Runtime.InteropServices	Native code support	mscorlib
System.Data	Access to DBMS	System.Data
System.Xml	Access to arbitrary data	System.Xml
System.Web	Server-side HTTP	System.Web
System.Diagnostics	Assertion/tracing	System.Diagnostics

Figure 1.6: Base class library

While it is possible to access the underlying platform features from a CLR-based executable, it is discouraged. Rather, programmers are encouraged to use the services of CLR-based libraries wherever possible. This is to prevent platform-dependencies from creeping into CLR-based applications, which ultimately would tie a program to Windows or Windows NT. In this respect, Windows now plays the role of the HAL from Windows NT, with **mscorlib** playing the role of Win32 by providing a layer of insulation from the underlying platform. Programs that only access the platform indirectly via CLR-based services are much more likely to be portable to other platforms that also support the CLR. This concept is illustrated in figure 1.7.

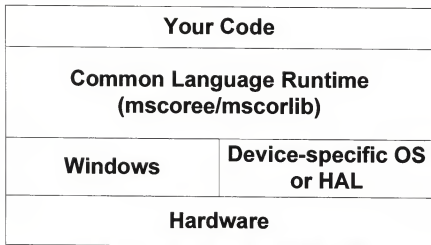


Figure 1.7: You and the Common Language Runtime

It is difficult to talk about the CLR without discussing the difference between a specification and an implementation. As part of the .NET initiative, Microsoft has submitted large parts of the platform to various standards organizations. In particular, Microsoft has submitted the Common Language Infrastructure (CLI) to the European Computer Manufacturer's Association (ECMA). The CLI includes the common type system and common intermediate language (CIL) used by the CLR. However, the CLR itself is not part of the ECMA submission. Rather, the CLR is an implementation of the CLI that is owned and controlled exclusively by Microsoft. In general, this book will not distinguish between the CLI specification and the CLR. However, when implementation-specific facets of the CLR are discussed, that will be explicitly noted.

The role of programming languages

Building CLR-based code requires a new compiler

- Language choice largely a matter of upbringing
- Programming model, libraries, type system shared by all languages
- Visual Studio.NET provides common RAD environment
- Command-line compilers (`csc.exe` and `vbc.exe`) for MAKEFILE wonks

It is difficult to discuss the Common Language Runtime without addressing programming languages. In general, the CLR supports any programming language that has a compiler that emits CLR executables. Programming languages are like flavors of ice cream in that what attracts a person to one language may repulse another person's esthetic sense. To that end, this book will avoid language-specific discussion whenever possible. Unfortunately, some programming language must be used to demonstrate various facets and features of the CLR. Examples in this book typically use C#. C# was chosen by the author simply because it is the de facto standard language for the platform, as is evidenced by numerous support tools, documentation and SDK samples. Note that while C# has its own unique syntax that is derived from C, C++ and Java, C# is ultimately just a programming language that imposes its own set of conventions and constructs over the underlying CLR.

CLR-based components can be programmed in any language with a supporting compiler or interpreter. Figure 1.8 shows the trade-offs between the five programming languages explicitly supported by Microsoft. Perhaps the most important distinction is the support (or lack of support) for CodeDOM compilation. CodeDOM compilation allows the language to be compiled on the deployment machine as well as the development machine. CodeDOM support is critical for a .NET language as it allows much more deployment flexibility, especially in ASP.NET. For that reason, non-CodeDOM languages are largely ignored throughout this text. While JScript is a viable choice, the vast majority of .NET development will take place in either C# or VB.NET. The choice between VB.NET and C# is largely personal, as their feature set is 90% identical. VB.NET is case-insensitive, C# is case-sensitive. C# supports C-style pointers, VB.NET does not. VB.NET supports untyped variables (a.k.a. late binding), C# does not. VB.NET auto-initializes local variables, C# does not. C# supports jagged arrays, VB.NET does not (easily at least).

Feature	VB.NET	JScript	C#	C++	ILASM
Compiler	VBC.EXE	JSC.EXE	CSC.EXE	CLEXE	ILASM.EXE
CodeDOM support	Yes	Yes	Yes	No	No
Dynamic property addition	No	Yes	No	No	No
Late binding	Automatic	Automatic	Manual	Manual	Manual
User-defined value types	Yes	No	Yes	Yes	Yes
Case-sensitive	No	Yes	Yes	Yes	Yes
Unsigned integer support	No	Yes	Yes	Yes	Yes
Method overloading	Yes	Yes	Yes	Yes	Yes
Operator overloading	No	No	Yes	Yes	N/A
C-style pointer support	No	No	Yes	Yes	Yes
Unmanaged type support	No	No	No	Yes	Yes
Native code support	No	No	No	Yes	No
Templates/generics	No	No	No	Yes	No
Multiple inheritance	No	No	No	Yes	No

Figure 1.8: .NET language features

Building a program for the CLR requires a CLR-aware compiler. Both C# and VB.NET provide command-line compilers that are freely available over the Internet. Figure 1.9 shows the simplest C# program, and figure 1.10 shows the same program in VB.NET. In general, programs that must stand the test of time are built using command-line tools such as the C# and VB.NET compilers (`csc.exe` and `vbc.exe`) as well as `nmake.exe` or Apache's ANT to automate build dependencies. Figure 1.11 shows a simple build process for a VB.NET program that uses a C#-based component. Note that the `/r` command-line switch is used to reference the component written in C#. Figure 1.12 shows the MAKEFILE that would automate this build procedure.

```
public class AnyNameYouWant {
    public static void Main() {
        System.Console.WriteLine("Hello, World");
    }
}
```

Figure 1.9: Hello, World - C#

```
public class AnyNameYouWant '{  
    public shared sub Main() '{  
        System.Console.WriteLine("Hello, World");  
    end sub '  
end class '}
```

Figure 1.10: Hello, World - VB.NET

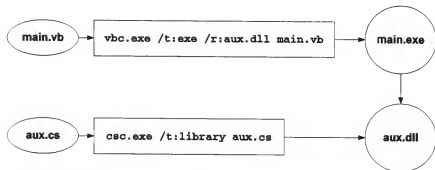


Figure 1.11: Building and using CLR components

```
all: main.exe  
  
main.exe : main.vb aux.dll  
    vbc.exe /t:exe /r:aux.dll main.vb  
  
aux.dll : aux.cs  
    csc.exe /t:library aux.cs
```

Figure 1.12: Sample MAKEFILE

The role of type

The Common Language Runtime is based on a pervasive type system

- Common managed type system for all programming languages
- Intra-component and inter-component type systems the same
- Single-rooted type system (System.Object) unlike COM, CORBA and Java
- Rich enough to support a wide array of paradigms (OOP/IBP/AOP/Functional/Imperative)
- Most services keyed off of type

Integration technologies such as COM or CORBA tended to have a type system that was disjoint from the programming language used within a component. This is illustrated in figure 1.13. Having to deal with two type systems meant that a non-trivial amount of effort was spent converting between the one type system (e.g., your programming language's type system) and another (e.g., COM's type system). While this allowed old programs to integrate "at the edges," it meant that new programs needed to straddle the fence if one was to really leverage the underlying technology.

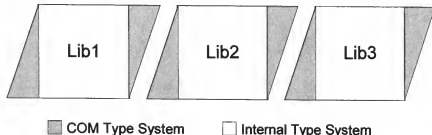


Figure 1.13: Type fragmentation under COM

Integration using the CLR is based on a common type system that permeates all component code. CLR-aware compilers translate programming language constructs into instructions and type definitions that the CLR can understand. This means that unlike COM or CORBA-based integration, there is no need for type system conversion at the edges of your component. This is illustrated in figure 1.14.

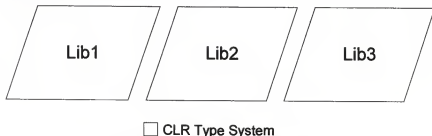


Figure 1.14: Pervasive Type in the CLR

Unlike technologies such as COM and CORBA, the CLR tends to work on all aspects of program execution, not just at integration boundaries. Along these lines, new compilers are needed for virtually all languages that target the CLR. To build or use CLR components, it does not matter which programming language/environment you use, provided it targets the CLR.

The CLR instruction set (a.k.a. common intermediate language or CIL) was designed to support a wide range of programming languages. One of the jobs of the CLR is to translate the CIL down to machine code prior to executing it. This translation is called Just-In-Time compiling or JITting. It is the job of the JITer to perform the processor-specific back-end optimizations that in the past were typically done on the developer's machine.

Common type system used by the CLR is shown in figure 1.15. In the CLR, every type is based on this common type system. In particular, all types are compatible with a universal type known as `System.Object`. That means that any field, variable, parameter, literal, or expression can be assigned to variables of type `System.Object`.

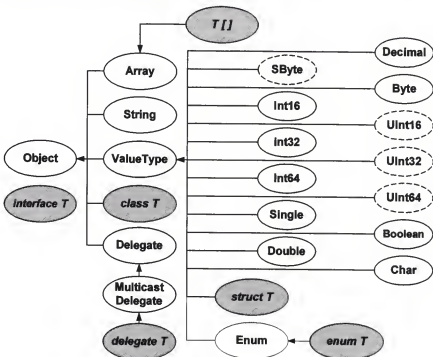


Figure 1.15: The CLR type system

In the CLR, every object and value is an instance of a type. This means that one can deal with objects, values and literals uniformly, including invoking methods on a value that is not formally considered an object. A simple program that demonstrates this is shown in figure 1.16.

```
public static void TypeFun() {  
    int x = 3;  
    System.Int32 y = 4; // x and y are same type  
    System.String s = x.ToString();  
    s = (3 - 2).ToString();  
    System.Object o = x;  
    s = o.ToString();  
}
```

Figure 1.16: Using the type system

CLS == VB and script

The Common Language Specification (CLS) subsets the universally supported types

- CLS prohibits unsigned integral types
- CLS prohibits pointer types and jagged arrays
- CLS prohibits overloading based on case or member kind
- Language-specific keywords must be "escapable"

The CLR type system is broad enough to support a wide range of programming language features. However, relying on these features at the edges of your component assumes that the consuming application or component is written in a language that can handle these features. Realistically, not all programming languages support the same set of features (or subsequently, types). To ensure smooth integration across languages, Microsoft defined the Common Language Specification (CLS) to identify the subset of the CLR type system that all languages should support. This does not mean that you are in any way discouraged from using all of the available type system within your component. Rather, it simply states that you should ensure that the types and methods used at the edges of your component are CLS-compliant (as shown in figure 1.17). Developers working in languages with richer type support can use the `[System.CLSCompliant]` attribute to tell their compiler to enforce CLS compliance for all public members. As shown in figure 1.18 this attribute causes non-CLS compliant public members to be flagged as compilation errors. Note that non-public members (in this example, the method `c`) may use non-CLS compliant parameters with no restriction. Also, the `[System.CLSCompliant]` attribute may be used to turn off CLS enforcement on a member-by-member basis, as is the case with method `d`.

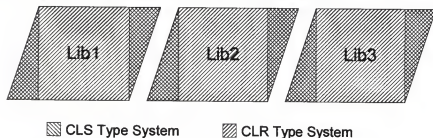


Figure 1.17: Type subsetting with the Common Language Specification

```
using System;
[assembly: CLSCompliant(true)]

public class MyClass {
    // OK, all parameters CLS compliant
    public void a(int x) { }
    // Error, public member using non-CLS compliant parameter
    public void b(uint x) { }
    // OK, not a public member
    internal void c(uint x) { }
    // OK, CLSCompliant attribute used
    [CLSCompliant(false)] public void d(uint x) { }
}
```

Figure 1.18: CLS compliance

The two primary environments that require CLS compliance are VB.NET and scripting languages. In particular, VB.NET cannot easily deal with unsigned integral types, non-rectangular arrays, or member/type names that differ only by case. Additionally, pointer types are not allowed by the CLS despite the fact that the CLR type system, C++, and C# are perfectly happy to support them.



Execution scope and the CLR

Running applications are modeled in the CLR as AppDomains

- AppDomains fill the role of the OS process in the CLR
- An AppDomain is scoped to a particular process/runtime
- A process/runtime can host multiple AppDomains
- AppDomains are cheaper than OS processes
- An OS thread can switch AppDomains much faster than a process switch
- An object's type controls cross-appdomain marshaling

AppDomains fill many of the same roles filled by an operating system process. AppDomains, like processes, scope the execution of code. AppDomains, like processes, provide a degree of fault isolation. AppDomains, like processes, provide a degree of security isolation. AppDomains, like processes, own resources on behalf of the programs they execute. In general, most of what you may know about an operating system process probably applies to AppDomains.

AppDomains are strikingly similar to processes, yet they are ultimately two different things. A process is an abstraction created by your operating system. An AppDomain is an abstraction created by the CLR. While a given AppDomain resides in exactly one OS process, a given OS process can host multiple AppDomains. This relationship is shown in figure 1.19.

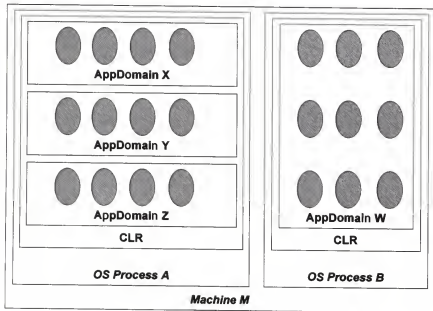
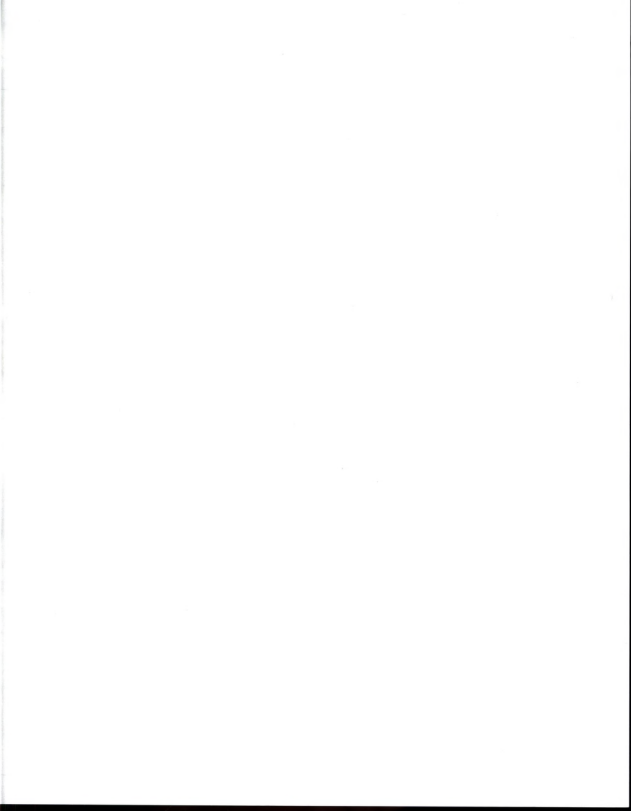


Figure 1.19: Objects, AppDomains, and Processes

To a managed application, the AppDomain defines the edges of your execution boundary. Code in another appdomain cannot directly access your objects. Similarly, your code cannot directly manipulate objects that reside in another AppDomain. To work with objects that reside in another AppDomain, a proxy to the remote object must be obtained by some means and used to indirectly manipulate the target object. This is in keeping with the analogy to operating system processes. Code in one process cannot directly manipulate objects in another process - a proxy mechanism of some sort is required. That said, it is much less expensive to cross an AppDomain boundary between two AppDomains that are hosted in the same process, than it is to cross an actual process

boundary. So AppDomains provide the same isolationary benefits that processes do, but without the same degree of performance degradation.



The role of ASP.NET

A significant amount of code will be compiled and executed by ASP.NET

- Most Internet-scale servers/services are exposed via HTTP
- Most N-tier architectures make heavy use of HTTP
- Most .NET user-interfaces will be delivered via the web
- ASP.NET has its own integrated compilation and execution environment
- ASP.NET uses AppDomain-per-virtual directory for isolation and performance

While the initial motivation for the CLR was to address problems in COM, there is no doubt that many developers will rely on ASP.NET as their development platform. ASP.NET uses the CLR to replace the existing ISAPI/ASP infrastructure of IIS with a more efficient and easier-to-use framework for servicing HTTP requests.

The `System.Web` library provides the support for server-side HTTP programming in ASP.NET. `System.Web` allows a CLR-based type to be bound to a particular URL or URL pattern. When an incoming request is received by ASP.NET, the `System.Web` infrastructure identifies which type should be loaded and forwards the request to an instance of that type. `System.Web` provides a rich set of services for server-based code, including process and state management, authentication, declarative authorization controls, and request processing. One of the more interesting aspects of ASP.NET is that it allows code to be deployed either in binary form or in source form.

ASP.NET always executes compiled code, even when the code is deployed in source form. This requires the compiler to exist on the server machine, not the development machine (although it is hard to imagine a development machine not having a compiler). ASP.NET uses file change notifications to determine when a source file needs to be recompiled to produce a new DLL. ASP.NET only invokes the C# (or VB or JScript) compilers when the cached DLL is older than the modification date of the source file(s) used to produce it. In general, this means that the first request after a source file change will trigger a "build," but that all subsequent requests will execute from the cached DLL. This architecture is very similar to that of Java Server Pages (JSP).

ASP.NET provides a variety of ways to format the response to the request. The two most common ways are (a) to write an exemplar of the response in the ASPX file and sprinkle it with inline-code fragments and (b) to simply call into an external CLR class to write out the response using the `System.Web.HttpContext` class. Figure 1.20 shows an example of the former. Figure 1.21 shows an example of the latter.

```
<%@page language="C#" %>
Hello <%= "Wor" + "ld" %>
```

Figure 1.20: Hello, World - C#/ASP.NET

```
<%@ webhandler language="C#" class="AnyNameYouWant" %>
using System.Web;

public class AnyNameYouWant : IHttpHandler {
    public void ProcessRequest(HttpContext call) {
        call.Response.Write("Hello, World");
    }
    public bool IsReusable { get { return true; } }
}
```

Figure 1.21: HelloWorld.ASHX - Hello, World in C#/ASP.NET (revisited)

In ASP.NET, many of the options usually specified in a MAKEFILE are specified either in configuration files or in the source files themselves. As shown in figure 1.22, the ASP.NET infrastructure parses .ASPX files to determine which code should execute based on a specific client request. While it is possible to insert executable code inside of the .ASPX file, it is generally considered better style to put the code in a separate file "behind" the ASPX file, which usually contains HTML or XML-based content.

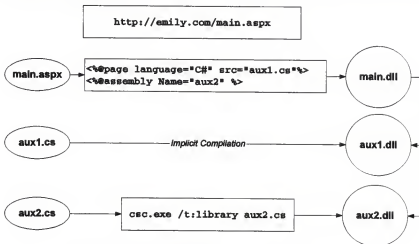
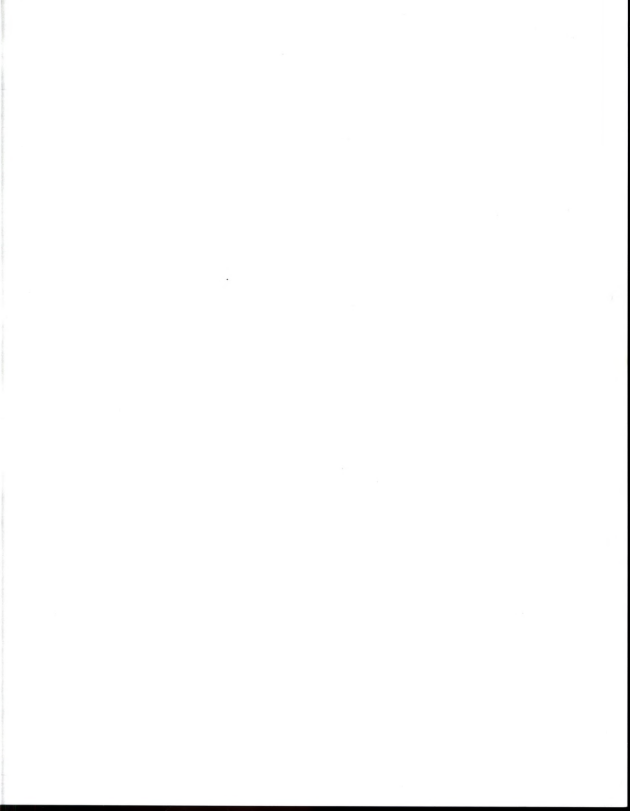


Figure 1.22: Building and using ASP.NET components

Unlike the previous generation of ASP, which used separate processes to provide isolation boundaries for web applications, ASP.NET uses a single worker process (ASPNET_WP.EXE) that contains a separate AppDomain for each virtual directory in the IIS metabase. This allows ASP.NET to offer the same level of isolation between web applications that are potentially hosting code written by different customers without incurring the cost of creating and destroying separate process address spaces.



The role of XML, SOAP and Web Services

XML is the future of component integration

- XML is supported by every major (and minor) system
- XML has a type system based on XML Schemas (XSD)
- XML has a messaging protocol based on SOAP/XP
- Web Services are just a fancy term for XML/HTTP-based RPC server

The CLR is a great way to integrate software in a single OS process. However, the CLR cannot span the Internet, so some other technology needs to be used to integrate software across machine boundaries. Enter web services.

A web service is a software component or program that is accessible to other programs over Internet-friendly protocols, typically HTTP and XML. HTTP is used because it is often the only protocol that can traverse the proxy and firewall infrastructure of most organizations. XML is used because it is a flexible data representation that can be adapted to a wide variety of applications. To that end, the trend in component software is to move away from vendor-specific communication protocols and technologies in favor of XML. XML provides a vendor-neutral way to externalize typed objects and values from one runtime environment (e.g., the CLR) to another (e.g., Java or Perl). XML also provides a type system of its own in the form of XML Schema.

In an ideal world, all information would remain in its XML form throughout the processing pipeline. However, the state of the practice at the time of this writing is to convert XML-based information into a traditional programmatic type system. Figure 1.23 shows this approach to using XML as a component integration technology. Note that in this approach, the XML schema type system is used to bridge the type systems of the components themselves (e.g., CLR, Java). Yes, this approach looks strikingly similar to the approaches used by COM and CORBA in the 1990's. However, until programming languages become better at dealing with XML types as first-class citizens, this approach will likely dominate distributed application development for some time.

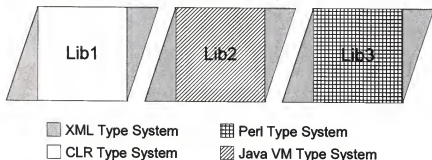


Figure 1.23: Type segregation under XML

Summary

- The CLR is an evolutionary step in component software
- The CLR is based on a pervasive type system
- The CLR provides extensible descriptions of all possible types during program execution
- The CLS identifies the subset of the type system that should be supported by most/all languages
- ASP.NET provides an HTTP-based server-side development and execution framework using the CLR
- Web Services incorporate XML and HTTP to support remote operation invocation

Module 2

Assemblies and Modules

The Common Language Runtime requires all types to belong to an assembly. Assemblies consist of one or more executable files called modules, each of which consists of code and metadata. Assemblies act as the smallest distribution unit for component code in the .NET Framework and represent both encapsulation and versioning boundaries.

After completing this module, you should be able to:

- ❑ understand how code gets loaded by the Common Language Runtime
- ❑ effectively manage public key-based assembly names
- ❑ create, install, and manage assemblies
- ❑ manage side-by-side installations of the same component

Assemblies and the .NET Framework

The Common Language Runtime requires all types to belong to an assembly. Assemblies act as the smallest distribution unit for component code in the .NET Framework.

Modules defined

A module is a byte-stream that contains metadata, code, and resources

- Modules typically stored as files in the filesystem or web
- Metadata contains type definitions
- Code contains method implementations
- Resources contain strings and other static data
- Modules are extended PE/COFF executables
- Modules typically use .NETMODULE, .DLL, or .EXE extensions
- Raw modules produced using CSC.EXE's /t:module switch

The term "component" has been one of the most ill-defined terms in the history of software engineering. The Component Object Model (COM) itself never had a firm definition of the term, despite the word component appearing prominently in the technology name. In this great tradition, the heir apparent to COM, the CLR, does not attempt to put any more shape around the term than its predecessor. That stated, if one adheres to Clemens Szyperksi's definition of the term, then the CLR version of a component is called an assembly. As this chapter describes, assemblies act as logical collections of modules, which is where code and type definitions ultimately reside.

Programs written for the CLR reside in modules. A CLR module is a byte-stream, typically stored as a file in the local file system or on a web server. As shown in figure 2.1, a CLR module uses an extended version of the PE/COFF executable file format. By extending the PE/COFF format rather than starting from scratch, CLR modules are also valid Win32 modules that can be loaded using the `LoadLibrary` system call. However, very little PE/COFF functionality is actually used by a CLR module. Rather, the majority of a CLR module's contents are stored as opaque data in the `.text` section of the PE/COFF file.

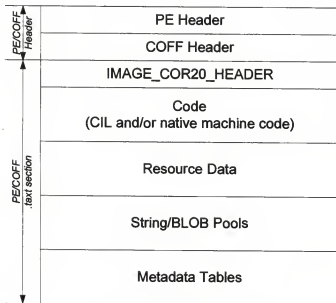


Figure 2.1: CLR Module Format

CLR modules contain code, metadata, and resources. The code is typically stored in common intermediate language (CIL) format, although it may also be

stored as processor-specific machine instructions. The module's metadata describes the types defined in the module, including names, inheritance relationships, method signatures, and dependency information. The module's resources consist of static read-only data such as strings, bitmaps, and other aspects of the program that are not stored as executable code.

The file format used by CLR modules is fairly well documented, however, few developers will ever encounter the format in the raw. Even developers who need to generate programs on the fly will typically use one of the two facilities provided by the CLR for programmatically generating modules. The `IMetaDataEmit` interface is a low-level COM interface that can be used to generate modules programmatically from classic C++. The `System.Reflection.Emit` namespace is a higher-level library that can be used to generate modules programmatically from any CLR-friendly language (e.g., C#, VB.NET). For the vast majority of developers who simply need to generate code during development, not at runtime, a CLR-friendly compiler will suffice.

The C# compiler (`CSC.EXE`) and VB.NET compiler (`VBC.EXE`) both translate source code into CLR modules. The two compilers accept a common set of command-line switches to control which kind of module to produce. The `/target` command-line switch (or its shortcut `/t`) supports four possible options: `/t:module` produces a "raw" module that by default will use the `.netmodule` file extension. Modules in this format cannot be deployed by themselves as stand-alone code, nor can they be loaded directly by the CLR. Rather, "raw" modules must be associated with a full-fledged component (called an assembly) prior to deployment. In contrast, compiling with the `/t:library` option produces a module that contains additional metadata that allows it to be deployed as stand-alone code.

A module produced by compiling with `/t:library` will have a `.DLL` file extension by default. Such a module can be loaded directly by the CLR but cannot be launched as an executable program from a command shell or the Windows Explorer. To produce this kind of module, you must compile with either the `/t:exe` or `/t:winexe` options. Both options produce a file whose extension is `.EXE`. The only difference between these two options is that the former assumes the console UI subsystem, the latter option assumes the GUI subsystem. If no `/t` option is specified, the default is `/t:exe`.

Modules produced using either the `/t:exe` or `/t:winexe` options must have a main entry point defined. The main entry point is the method that the CLR will execute automatically when the program is launched. The method must be declared as `static` and, in C# or VB.NET, must be named `Main`. The entry point method can be declared to return no value or to return an `int` as its exit code. The entry point method can be declared to accept no parameters, or it can be declared to accept an array of strings, which will contain the parsed

command-line arguments from the shell. The following are four legal implementations for the `Main` method in C#:

```
static void Main() { }  
static void Main(string[] argv) { }  
static int Main() { return 0; }  
static int Main(string[] argv) { return 0; }
```

Note that these methods do not need to be declared `public`. The `Main` method does, however, need to be declared inside of a class definition, although the name of the class is immaterial.

The following is minimal C# program that does nothing but print the string `Hello, World` to the console:

```
class myapp {  
    static void Main() {  
        System.Console.WriteLine("Hello, World");  
    }  
}
```

In this example, there is exactly one class with a static method called `Main`. Presenting the C# or VB.NET compilers with source files containing more than one type with a static method called `Main` is an error due to the ambiguity. To resolve this ambiguity, the `/main` command-line switch can be used to tell the C# or VB.NET compiler which type to use for the program's initial entry point.

Assemblies defined

Assemblies are logical collections of one or more modules

- Assemblies are the atom of deployment in the CLR
- Type names are scoped by their containing assembly
- Types/code loaded based on containing assembly, not module
- Only types and members marked `public` are visible outside the assembly
- `internal` members and types inaccessible outside of assembly
- `private` members inaccessible outside of their declaring type

An assembly is a named collection of one or more modules. As just described, modules are physical constructs that exist as byte-arrays, typically in the file system. Assemblies are logical constructs and are referenced by location-independent names that must be translated to physical paths either in the file system or on the Internet. Those physical paths ultimately point to one or more modules that contain the type definitions, code, and resources that make up the assembly.

The CLR allows assemblies to be partitioned into multiple modules primarily to support deferred loading of infrequently accessed code. This feature is especially useful when code download is used, as the initial module can be downloaded first, with secondary modules only being downloaded on an as-needed basis. Being able to build multi-module assemblies also enables mixed-language assemblies. This allows a developer to work in a high-productivity language (e.g., Logo.NET) for the majority of their work but to write low-level grunge code in a more flexible language (e.g., C++). By conjoining the two modules into a single assembly, the C++ and Logo.NET code can be referenced, deployed, and versioned as an atomic unit.

Though an assembly may consist of more than one module, a module is generally only affiliated with one assembly. As a point of interest, if two assemblies happen to reference a common module, the CLR will treat this as if there are two distinct modules, which results in two distinct copies of every type in the common module. For that reason, the remainder of this chapter will assume that a module is affiliated with exactly one assembly.

Assemblies are the atom of deployment in the CLR and are used to package, load, distribute, and version CLR modules. While an assembly may consist of multiple modules and auxiliary files, the assembly is named and versioned as an atomic unit. If one of the modules in an assembly must be versioned, then the entire assembly must be redeployed, as the version number is part of the assembly name, not the underlying module name.

Modules typically rely on types from other assemblies. At the very least, every module relies on the types defined in the `mscorlib` assembly. Every CLR module contains a list of assembly names that identify which assemblies are used by this module. These external assembly references use the logical name of the assembly, which contain no remnants of the underlying module names or locations. It is the job of the CLR to convert these logical assembly names into module pathnames at runtime, as is discussed later in this chapter.

To assist the CLR in finding the various pieces of an assembly, every assembly has exactly one module whose metadata contains the assembly manifest. The assembly manifest is an additional chunk of CLR metadata that acts as a directory of adjunct files that contain additional type definitions and code. Modules that contain an assembly manifest can be loaded directly by the CLR. Modules that lack an assembly manifest can only be loaded indirectly by first loading a module whose assembly manifest refers to the manifest-less

module. Figure 2.2 shows two modules: one with an assembly manifest and one without one. Note that of the four /t compiler options, only /t:module produces a module with no assembly manifest.

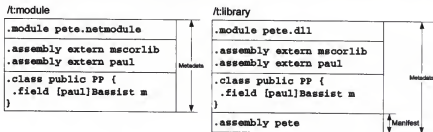


Figure 2.2: Modules and Assemblies

Figure 2.3 shows an application that uses a multi-module component and figure 2.4 shows the MAKEFILE that would produce it. In this example, `code.netmodule` is a module that does not contain an assembly manifest. To make it useful, one needs a second module (in this case `component.dll`) that provides an assembly manifest that references `code.netmodule` as a subordinate module. This is achieved using the /addmodule switch when compiling the containing assembly. Once this assembly is produced, the types defined in `component.dll` and `code.netmodule` all are scoped by the name of the assembly (`component`). Programs such as `application.exe` use the /r compiler switch to reference the module containing the assembly manifest. This makes the types in both modules available to the referencing program.

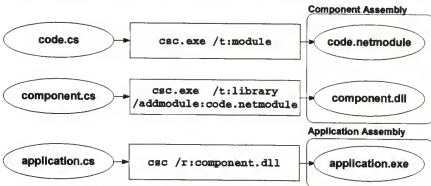


Figure 2.3: Multi-module assemblies using CSC.EXE

```
# code.netmodule cannot be loaded as is until an assembly
# is created
code.netmodule : submodule.cs
    csc /t:module submodule.cs

# types in component.cs can see internal and public members
# and types defined in submodule.cs
component.dll : component.cs code.netmodule
    csc /t:library /addmodule:code.netmodule component.cs

# types in application.cs cannot see internal members and
# types defined in submodule.cs (or component.cs)
application.exe : application.cs component.dll
    csc /t:exe /r:component.dll application.cs
```

Figure 2.4: Multi-module assemblies using CSC.EXE and NMAKE

The assembly manifest resides in exactly one module and contains all of the information needed to locate types and resources defined as part of the assembly. Figure 2.5 shows a set of modules composed into a single assembly as well as the CSC.EXE switches required to build them. Notice that in this example, the assembly manifest contains a list of file references to the subordinate modules `pete.netmodule` and `george.netmodule`. In addition to these file references, each of the public types in these subordinate modules is listed using the `.class extern` directive, which allows the list of public types to be discovered without traversing the metadata for each of the modules in the assembly. Each entry in this list specifies both the file name that contains the type as well as the numeric metadata token that uniquely identifies the type within its module. Finally, the module containing the assembly manifest will contain the master list of externally referenced assemblies. This list consists of the dependencies of every module in the assembly, not just the dependencies of the current module.

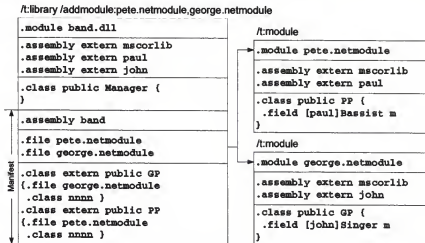


Figure 2.5: A multi-module assembly

Assemblies form an encapsulation boundary to protect internal implementation details from inter-assembly access. This protection can be applied to members of a type (e.g., fields, methods, constructors) or to a type as a whole. Marking a member or type as `internal` causes it to be available only to modules that are part of the current assembly. Marking a type or member as `public` causes it to be available to all code (both inside and outside the current assembly). Individual members of a type (e.g., methods, fields, constructors) may also be marked as `private`, which restricts access to only methods and constructors of the declaring type. This supports classic C++-style programming where intra-component encapsulation is desired. In a similar vein, members of a type can be marked `protected`, which broadens the access allowed by `private` to include methods and constructors of derived types. The `protected` and `internal` access modifiers may be combined, which provides access to types that are either derived from the current type or are in the same assembly as the current type.

Assemblies scope the type definitions of a component. CLR types are uniquely identified by their assembly name/type name pair. This allows two definitions of the type `Customer` to coexist inside the runtime without ambiguity, provided that they are each affiliated with different assemblies. While it is possible for multiple assemblies to define the type `Customer` without confusing the runtime, it does not help the programmer who wants to use two or more definitions of the same type name in a single program, since the symbolic type name is always `Customer` no matter which assembly defines it. To address this limitation that exists in most programming languages, CLR type names may have a namespace prefix. This prefix is a string that typically

begins with either the organization name of the developer (e.g., Microsoft, DevelopMentor) or `System` if it is part of the .NET framework. An emerging convention is to name the assembly based on the namespace prefix. For example, the .NET XML stack is deployed in the `System.Xml` assembly, and all of the contained types use the `System.Xml` namespace prefix. This is simply a convention and not a rule. For example, the type `System.Object` resides in an assembly called `mscorlib`, not in the assembly called `System`, even though there actually is an assembly called `System`.

Assembly names

All assemblies have a four-part name that uniquely identifies the locale and developer of the component

- The simple Name typically corresponds to the file name (no extension)
- The Version identifies the major/minor/build/revision numbers
- The CultureInfo corresponds to language and region
- The PublicKey identifies the developer
- Display names are stringified assembly names suitable for human entry
- Note: Namespace prefixes of types may or may not match the Name of the assembly

All assemblies have a four-part name that uniquely identifies the friendly name, locale, developer, and version of the component. These names are stored in the assembly manifest of both the assembly itself as well as all assemblies that reference it. The four-part assembly name is used by the assembly resolver to find the correct component at load-time. The CLR provides programmatic access to assembly names via the `System.Reflection.AssemblyName` type.

The `Name` property of the assembly name typically corresponds to the underlying file name of the assembly manifest sans any extension that may be in use. This is the only part of the assembly name that is not optional. In simple scenarios, this is all that is needed to locate the correct component at load-time. When building an assembly, this part of the name is automatically selected by your compiler based on the target file name. While strictly speaking, the `Name` of the assembly does not need to match the underlying file name, keeping the two in sync makes the job of the assembly resolver (and system administrators) much simpler.

All assembly names have a four-part version number (`Version`) of the form `Major.Minor.Build.Revision`. If you do not set this version number explicitly, its default value will be 0.0.0.0. The version number is set at build-time using a custom attribute in the source code or using a command line switch. The `System.Reflection.AssemblyVersion` attribute accepts a variety of string formats as shown in figure 2.6. When specifying the version number, the `Major` version number is mandatory. Any missing components are assumed to be zero. At build-time, the `Revision` is allowed to be specified as `*`, which causes the compiler to use the wall clock to produce a unique, monotonically increasing revision number for each compilation. If a `*` is specified for the `Build` number, the number emitted into the assembly manifest is based on the number of days that have elapsed since February 1, 2000, ensuring that each day has its own unique build number but that a given build number "expires" after 24 hours. The values of `Major` and `Minor` must be explicitly specified. How the `Version` of the assembly is used by the assembly loader and resolver is discussed later in this chapter.

Attribute Parameter	Actual Value
1	1.0.0.0
1.2	1.2.0.0
1.2.3	1.2.3.0
1.2.3.4	1.2.3.4
1.2.*	1.2.d.s
1.2.3.*	1.2.3.s
<absent>	0.0.0.0

** where d is the number of days since Feb. 1, 2000 and s is the number of seconds since midnight /2*

Figure 2.6: Inside the AssemblyVersion attribute

Assembly names can contain a `CultureInfo` that identifies the spoken language and country code that the component has been developed for. The `CultureInfo` is specified using the `System.Reflection.AssemblyCulture` attribute, which accepts a two-part string as specified by IETF RFC 1766. The first part of the string identifies the spoken language using a two-character lower-case code. The (optional) second part of the string identifies the geographic region using a two-character upper-case code. The string "en-US" identifies U.S. English. Assemblies with a `CultureInfo` cannot contain code - rather, they must be resource-only assemblies (also known as satellite assemblies) that can only contain localized strings and other user-interface elements. Satellite assemblies allow a single DLL containing code to selectively load (and/or download) localized resources based on where they are deployed. Assemblies containing code (that is, the vast majority of assemblies) are said to be culture-neutral.

Finally, an assembly name can contain a public key (token) that identifies the developer of the component. An assembly reference may use either the full 128 byte public key or the 8 byte public key token. The public key (token) is used to resolve file name collisions between organizations, allowing as multiple `utilities.dll` components to coexist in memory and on disk provided that each one originates from a different developer, each of whom is guaranteed to have a unique public key. Public key management is discussed in detail in the next section.

Because assembly references occasionally must be entered by hand (for example, for use in configuration files), the CLR defines a standard format for writing 4-part assembly names as strings. This format is known as the display name of the assembly. The display name of the assembly always begins with the simple Name of the assembly, and is followed by an optional list of comma-delimited properties that correspond to the other three properties of the

assembly name. If all four parts of the name are specified, the corresponding assembly reference is called a fully qualified reference. If one or more of the properties is missing, the reference is called a partially qualified reference. Figure 2.7 shows a display name and the corresponding CLR attributes used to control each property. Note that if an assembly with no culture is desired, the display name must indicate this using `Culture=neutral`. Also, if an assembly with no public key is desired, the display name must indicate this using `PublicKeyToken=null`. Both of these are substantially different from a display name with no `Culture` or `PublicKeyToken` property. Simply omitting these properties from the display name results in a partially-specified name that allows any `Culture` or `PublicKeyToken` to be matched. In general, you should avoid using partially-specified assembly names; otherwise, various parts of the CLR will work in unexpected (and unpleasant) ways.

Display Name of Assembly Reference

`yourcode, Version=1.2.3.4, Culture=en-US, PublicKeyToken=1234123412341234`

or *neutral*

or *null*

C# Code

```
using System.Reflection;
[assembly: AssemblyVersion("1.2.3.4")]
[assembly: AssemblyCulture("en-US")] // resource-only assem
[assembly: AssemblyKeyFile("acmecorp.snk")]
```

Figure 2.7: Fully specified assembly names

Public Keys and Assemblies

Assemblies use public-key technology both to identify the developer and to prevent tampering

- Cryptographic signature based on public/private key pair
- Cannot tamper with assembly without resigning
- Originator in target assembly contains complete 1024 bit RSA public key
- Originator in assembly reference contains 64 bit hash of full public key
- The `SN.EXE` tool manages public/private key files
- The `AssemblyKeyFile` and `AssemblyDelaySign` attributes associate keys with assemblies

The CLR uses public key technology to both uniquely identify the developer of a component as well as to protect the component from being tampered with once it is out of the original developer's hands. Each assembly can have a public key embedded in its manifest that identifies the developer. Assemblies with public keys also have a digital signature that is generated before the assembly is first shipped that provides a secure hash of the assembly manifest, which itself contains hashes of all subordinate modules files. This ensures that once the assembly ships, no one can modify the code or other resources contained in the assembly. This digital signature can be verified using only the public key, however, the signature can only be generated with the corresponding private key, which savvy developers guard more closely than their source code. The current builds of the CLR uses RSA public/private keys and SHA hashing to produce the digital signature. While the public key used to sign the assembly is a unique fingerprint for each developer, it does not provide the same level of non-repudiation that digital certificates provide. For example, there is no way to look up the developer's identity based solely on an assembly's public key. The CLR does provide support for embedding digital certificates into assemblies, but that is outside the scope of this chapter.

The .NET SDK ships with a tool (`SN.EXE`) that simplifies working with public and private keys during development and deployment. Running `SN.EXE` with the `-k` option creates a new file that contains a newly generated public/private key pair. This file contains your private key, so it is critical that you practice safe computing and not leave this file in a non-secured location. Because the private key is so critical, most organizations postpone the actual signing of the assembly until just before shipping. To allow all developers in an organization to access the public key without having access to the private key, `SN.EXE` supports stripping out the public key portion using the `-p` option. This option creates a new file that contains only the public key. The conventional file extension for both public and public/private key files is `.SNK`.

The public key produced by `sn.exe` is a 128 byte blob with an additional 32 bytes of header information. To keep the size of assembly references (and their display names) compact, an assembly reference can use a public key token, which is an 8 byte hash of the full public key. The assembly references emitted by most compilers use this token in lieu of the full public key to keep the overall size of the manifest small. You can calculate the token for a public key by using `SN.EXE`'s `-t` or `-T` options. The former calculates the token based on a `.SNK` file containing only a public key. The latter calculates the token based on a public key stored in an assembly's manifest. Figure 2.8 shows the `SN.EXE` tool in action.

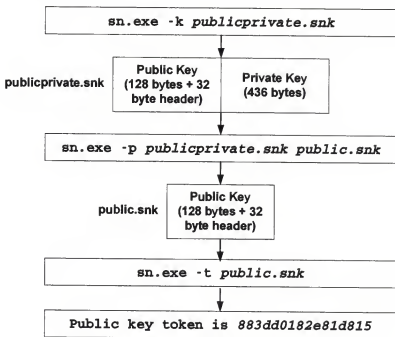


Figure 2.8: Managing public/private keys using SN.EXE

Development tools that support the CLR must provide some mechanism for developers to sign their assemblies, either via custom attributes or command-line switches. The `System.Reflection.AssemblyKeyFile` attribute tells the compiler where to find the .SNK file that contains the developer's public key. This attribute will work with either the public/private key pair or the public-only key, which allows developer's who are not trusted with their organization's private key to build, test, and debug their components. In order to build an assembly using only a public key, you must also use the `System.Reflection.AssemblyDelaySign` attribute to inform the compiler that no private key is present and that no meaningful digital signature can be produced. When delay signing is used, space is reserved for the digital signature so that a trusted member of the organization can resign the assembly without having to replicate the original developer's build environment. In general, assemblies that have a public key but do not have a valid signature cannot be loaded or executed. To allow delay-signed assemblies to be used during development, this policy can be disabled on an assembly-by-assembly basis using the `-vr` option to `SN.EXE`.

Figure 2.9 shows the `AssemblyKeyFile` attribute used from C#. This figure also shows the resultant assembly as well as another assembly that references

it. Note that the 128 byte public key is stored in the target's assembly manifest, along with a digital signature to protect the assembly from tampering. Also note that the second assembly that references the target only contains the 8 byte public key token. Because the target assembly was built with delay signing turned off, the assembly can now be deployed and loaded in secured environments. In contrast, the target assembly produced by the C# compiler shown in figure 2.10 is not suitable for deployment, as it is built with delay signing turned on. However, once a trusted individual signs the assembly with the full private key, the assembly is ready to be deployed. Note that in this example, the SN.EXE tool is used with the -R option, which overwrites the digital signature in the target assembly with one based on the public/private key provided on the command line. You can manually verify that an assembly has been signed using SN.EXE with the -v or -vf option. The latter overrides any configured settings that might disable signature verification.

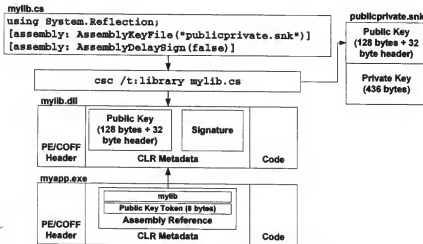


Figure 2.9: Strong assembly references

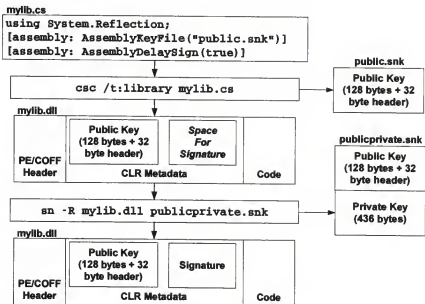
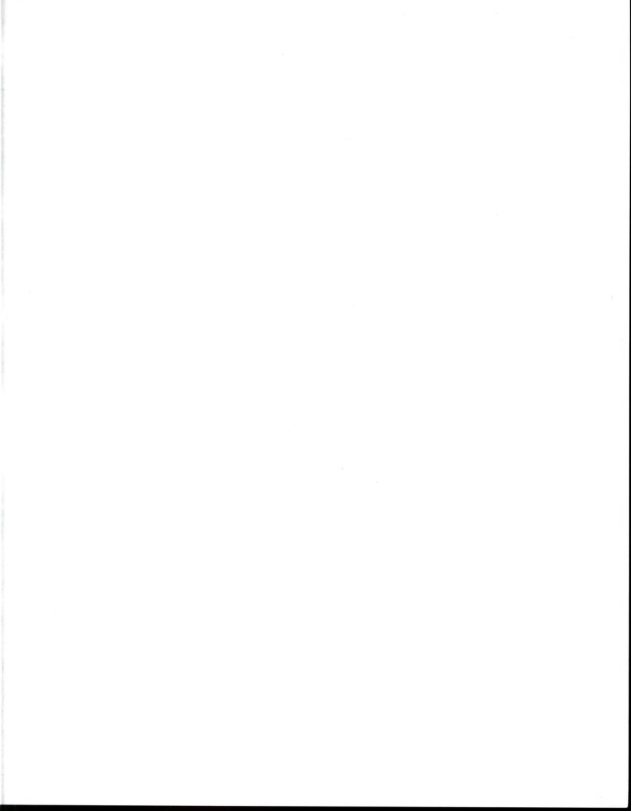


Figure 2.10: Delay signing an assembly



Loading and resolving assemblies

Assemblies can be loaded either by an explicit URL or a four-part assembly name

- `Assembly.LoadFrom` loads an assembly based on an explicit CODEBASE (e.g., a file name or URL)
- `Assembly.Load` first uses the assembly resolver to resolve a 4-part assembly name to a file prior to loading
- Subordinate assemblies always loaded using assembly resolver
- Subordinate modules and assemblies only loaded on demand
- CODEBASE hints specified using per-application/machine configuration files
- Assemblies from non-file URLs are cached in the download cache (watch out for `SecurityException` from untrusted code)

fuslogvw.exe - to debug loader

The CLR allows component code to be loaded either by a location-independent logical name or a physical path. Loading code based on its physical path is the job of the assembly loader. Loading code based on its logical name is the job of the assembly resolver. The assembly loader loads and initializes assemblies based on an explicitly specified location. The assembly loader accepts either a file name or URL and, assuming the corresponding file is a valid CLR assembly, loads the assembly into memory. If the specified file cannot be found, the loader will throw a `System.FileNotFoundException` exception. If the specified file can be found but is not a CLR assembly, the loader will throw a `System.BadImageFormatException` exception.

The CLR loader is responsible for loading and initializing assemblies, modules and types. The CLR loader loads and initializes as little as it can get away with. Unlike the Win32 loader, the subordinate modules (or assemblies) are not resolved and loaded automatically. Rather, the subordinate pieces are lazily loaded on demand only if they are actually needed. This not only speeds up program initialization time, but also reduces the amount of resources consumed by a running program.

Loading in the CLR is typically triggered by the JIT compiler based on types. When the JIT compiler tries to convert a method body from CIL to machine code, it needs access to the type definition of the declaring type as well as the type definition for the type's fields. Moreover, the JIT compiler also needs access to the type definitions used by any local variables or parameters of the method being JIT compiled. Loading a type implies loading both the assembly and module that contain the type definition.

This policy of loading types (and assemblies and modules) on demand means that parts of a program that are not used are never brought into memory. It also means that a running application will often see new assemblies and modules loaded over time as the types contained in those files are needed during execution. If this is not the behavior you want, you have two options. One is to simply declare hidden static fields of the types you want to guarantee are loaded when your type is loaded. The other is to interact with the loader explicitly.

The loader typically does its work implicitly on your behalf. Developers can interact with the loader explicitly via the assembly loader. The assembly loader is exposed to developers via the `LoadFrom` static method on the `System.Reflection.Assembly` class. This method accepts a CODEBASE string, which can be either a file system path or a URL. If the CODEBASE is a URL that uses any protocol other than `file:`, the caller must have `WebPermission` access rights or else a `System.SecurityException` exception is thrown. Additionally, assemblies at URLs with protocols other than `file:` are first downloaded to the download cache prior to being loaded.

Figure 2.11 shows a simple C# program that loads an assembly located at `file://C:/usr/bin/xyzzy.dll` and then creates an instance of the

contained type named `DevelopMentor.LOB.Customer`. In this example, all that is provided by the caller is the physical location of the assembly. When using the assembly loader in this fashion, the four-part assembly name is not used to locate the file, nor are any version policies applied.

```
using System;
using System.Reflection;
public class Utilities {
    public static Object LoadCustomerType() {
        Assembly a = Assembly.LoadFrom(
            "file://C:/usr/bin/xyzzy.dll");
        return a.CreateInstance("DevelopMentor.LOB.Customer");
    }
}
```

Figure 2.11: Loading an assembly with an explicit CODEBASE

While loading assemblies by URLs is somewhat interesting, most assemblies are loaded using assembly references that contain all or part of the four-part assembly name. When loading an assembly by name, the assembly resolver is used. The assembly resolver uses the assembly name to determine which underlying file to load into memory using the assembly loader. As shown in figure 2.12, this name-to-assembly resolution process takes into account a variety of factors, including the directory the application is hosted in, versioning policies, and other configuration details (all of which are discussed later in this chapter).

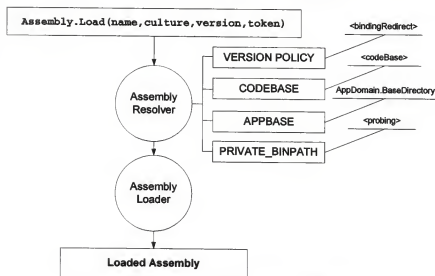


Figure 2.12: Assembly resolution and loading

The assembly resolver is exposed to developers via the Load method of the System.Reflection.Assembly class. As shown in figure 2.13, this method accepts a four-part assembly name (either as a string or as an AssemblyName reference) and superficially appears to be similar to the LoadFrom method exposed by the assembly loader. The similarity is only skin deep, as the Load method first uses the assembly resolver to find a suitable file using a fairly complex series of operations. The first of these operations is to apply a version policy to determine exactly which version of the desired assembly should be loaded.

```

using System;
using System.Reflection;

public class Utilities {
    public static Object LoadCustomerType() {
        Assembly a = Assembly.Load(
            "xyzy, Version=1.2.3.4, " +
            "Culture=neutral, PublicKeyToken=9a33f27632997fcc");
        return a.CreateInstance("DevelopMentor.LOB.Customer");
    }
}
  
```

Figure 2.13: Loading an assembly using the assembly resolver

Version policy

The assembly resolver can map the requested version of an assembly to a newer (or older) one via configured version policies

- Only applies to fully-specified assembly references (name, version, culture, publickey)
- Only applies when the assembly resolver is used
- Applied before any other actions are taken by the resolver
- Specified via configuration files per application and machine-wide

The assembly resolver begins its work by applying any version policies that may be in effect. Version policies are used to redirect the assembly resolver to load an alternate version of the requested assembly. A version policy can map one or more versions of a given assembly to a newer (or older) version, however, a version policy cannot redirect the resolver to an assembly whose name differs by any facet other than version number (i.e., an assembly named `Acme.HealthCare` cannot be redirected to an assembly named `Acme.Mortuary`). It is critical to note that version policies are only applied to assemblies that are fully specified by their four-part assembly name. If the assembly name is only partially specified (e.g., the public key token, version, or culture is missing), then no version policy will be applied. Also, no version policies are applied if the assembly resolver is bypassed by calling `Assembly.LoadFrom` directly, since you are only specifying a physical path, not an assembly name.

Version policies are specified via configuration files. There is a machine-wide configuration file and an application-specific configuration file. The machine-wide configuration file is always named `machine.config` and is located in the `%SystemRoot%\Microsoft.Net\Framework\Vx.y.z\CONFIG` directory. The application-specific configuration file is always located at the `APPBASE` for the application. For CLR-based .EXE programs, the `APPBASE` is the base URI (or directory) for the location the main executable was loaded from. For ASP.NET applications, the `APPBASE` is the root of the web application's virtual directory. The name of the configuration file for CLR-based .EXE programs is the same as the executable name with an additional ".config" suffix. For example, if the launching CLR program is in `C:\myapp\app.exe`, the corresponding configuration file would be `C:\myapp\app.exe.config`. For ASP.NET applications, the configuration file is always named `web.config`.

Configuration files are XML-based and always have a root element named `configuration`. Configuration files are used by the assembly resolver, the remoting infrastructure, and by ASP.NET. Figure 2.14 shows the basic schema for the elements used to configure the assembly resolver. All relevant elements are under the `assemblyBinding` element in the `urn:schemas-microsoft-com:asm.v1` namespace. There are application-wide settings to control probe paths and publisher version policy mode. Additionally, the `dependentAssembly` elements are used to specify version and location settings for each dependent assembly.

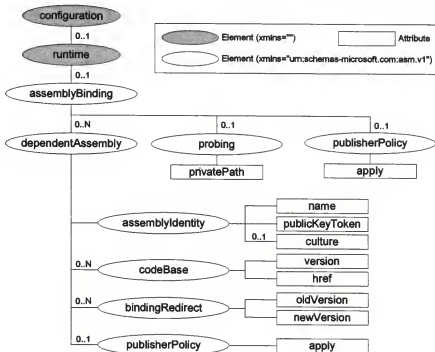


Figure 2.14: Assembly resolver configuration file format

Figure 2.15 shows a simple configuration file containing two version policies for one assembly. The first policy redirects version 1.2.3.4 of the specified assembly (Acme.HealthCare) to version 1.3.0.0. The second policy redirects versions 1.0.0.0 through 1.2.3.399 of that assembly to version 1.2.3.7. Note that if both the application's and machine's configuration files have a version policy for a given assembly, the application's policy is run first, and the resultant version number is then run through the machine-wide policy to get the actual version number used to locate the assembly. In this example, if the machine-wide configuration file had a version policy that redirected version 1.3.0.0 of Acme.HealthCare to version 2.0.0.0, the assembly resolver would use version 2.0.0.0 when version 1.2.3.4 was requested, since the application's version policy maps version 1.2.3.4 to 1.3.0.0 but the machine's version policy maps version 1.3.0.0 to version 2.0.0.0. Had the machine-wide policy instead mapped version 1.2.3.4 to version 2.0.0.0, the application would still use

version 1.3.0.0, even when version 1.2.3.4 is requested, simply because the application's version policy is applied first.

```
<?xml version="1.0" ?>

<configuration
  xmlns:asm="urn:schemas-microsoft-com:asm.v1"
>
  <runtime>
    <asm:assemblyBinding>
<!-- one dependentAssembly per unique assembly name -->
      <asm:dependentAssembly>
        <asm:assemblyIdentity
          name="Acme.HealthCare"
          publicKeyToken="38218fe715288aac" />
<!-- one bindingRedirect per redirection -->
        <asm:bindingRedirect oldVersion="1.2.3.4"
          newVersion="1.3.0.0" />
        <asm:bindingRedirect oldVersion="1-1.2.3.399"
          newVersion="1.2.3.7" />
      </asm:dependentAssembly>
    </asm:assemblyBinding>
  </runtime>
</configuration>
```

Figure 2.15: Setting the version policy

In addition to application-specific and machine-wide configuration settings, a given assembly can also have a publisher policy. A publisher policy is a statement from the component developer indicating which versions of a given component are compatible with one another. Publisher policies are stored as configuration files in the machine-wide global assembly cache. The structure of these files is identical to that of the application and machine configuration files. However, to be installed into the GAC, the publisher policy configuration file must be wrapped in a surrounding assembly DLL as a custom resource. Assuming that the file `foo.config` contains the publisher's configuration policy, the following command line would invoke the assembly linker (AL.EXE) and create a suitable publisher policy assembly for DevelopMentor.Code version 2.0:

```
al.exe /link:foo.config
      /out:policy.2.0.DevelopMentor.Code.dll
      /keyf:pubpriv.snk
      /v:2.0.0.0
```

The name of the publisher policy file follows the form `policy.major.minor.asmname.dll`. Because of this naming convention, a given assembly can only have one publisher policy file per major.minor version.

In this example, all requests for `DevelopMentor.Code` whose major.minor version is 2.0 will be routed through the policy file linked with `policy.2.0.DevelopMentor.Code.DLL`. If no such assembly exists in the GAC, then there is no publisher policy. As shown in figure 2.16, publisher policies are applied after the application-specific version policy but before the machine-wide version policy stored in `machine.config`.

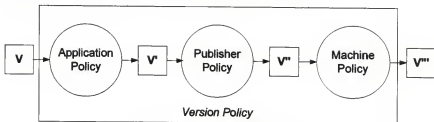


Figure 2.16: Version Policy

Given the fragility inherent in versioning component software, the CLR allows publisher version policies to be turned off on an application-wide basis using the `publisherPolicy` element in the application's configuration file. Figure 2.17 shows this element in a simple configuration file. When set to `apply="no"`, the publisher policies will be ignored for this application. When set to `apply="yes"` (or if not specified at all), the publisher policies will be used as just described. As shown in figure 2.14, the `publisherPolicy` element can enable/disable publisher policy on an application-wide or an assembly-by-assembly basis.

```
<?xml version="1.0" ?>

<configuration xmlns:rt="urn:schemas-microsoft-com:asm.v1">
  <runtime>
    <rt:assemblyBinding>
      <rt:publisherPolicy apply="no" />
    </rt:assemblyBinding>
  </runtime>
</configuration>
```

Figure 2.17: Setting the application to safe-mode



The assembly cache

Assemblies are first loaded from the machine-wide assembly cache

- The assembly cache makes assemblies available independent of where the application is located
- The download cache holds assemblies loaded from non-file-based URL
- The global assembly cache (GAC) holds system-level assemblies
- The assembly resolver always consults the GAC first
- The GAC is a secured resource and requires admin privileges to add/delete entries
- The GAC only contains signed assemblies with public keys

Once the assembly resolver decides which version of the assembly to load, it must then locate a suitable file to pass to the underlying assembly loader. The first place the CLR looks is in the directory specified by the `DEVPATH` OS environment variable. This environment variable is typically not set on the deployment machine. Rather it is intended for developer use only and exists to allow delay-signed assemblies to be loaded from a shared file-system directory. For that reason, the `DEVPATH` environment variable will not be discussed for the remainder of the chapter.

The first location that the assembly resolver uses to find an assembly is the Global Assembly Cache (GAC). The Global Assembly Cache is a machine-wide code cache that contains assemblies that have been installed for machine-wide use. The GAC allows administrators to install assemblies once per machine for all applications to use. Once the version policy has been applied, the assembly resolver looks in the Global Assembly Cache (GAC) to locate the requested assembly. To avoid system corruption, the GAC only accepts assemblies that have valid digital signatures and public keys. Additionally, entries in the GAC can only be deleted by administrators, which prevents non-admin users from deleting or moving critical system-level components.

The GAC is controlled by a system-level component (`FUSION.DLL`) that keeps a cache of DLLs under the `%WINNT%\Assembly` directory. `FUSION.DLL` manages this directory hierarchy for you and provides access to the stored files based on the 4-part assembly name as shown in figure 2.18. An Explorer shell extension provides a human-friendly view of the GAC, however, using a command-line shell, one can traverse the underlying directories and view the cached DLLs. Programmers rarely need to traverse this directory, as the tool `GACUTIL.EXE` provides a simple interface to the GAC. There is also a Windows Explorer shell extension that provides a user-friendly interface to the GAC.

Assembly Name	Version	Public Key Token	Mangled Subdirectory Beneath <windir>\assembly\gac\
abc	1.0.1.3	89abcde...	abc\1.0.1.3__89abcde_\abc.dll
abc	1.0.1.8	89abcde...	abc\1.0.1.8__89abcde_\abc.dll
xyz	1.1.0.0	89abcde...	Xyz\1.1.0.0__89abcde_\xyz.dll

Figure 2.18: Global Assembly Cache

Assembly resolving via CODEBASE or probing

If the assembly cannot be found in the GAC, the assembly resolver tries to use a CODEBASE hint to access the assembly

- Configuration files can/should provide a CODEBASE hint
- If matching file not accessible via provided CODEBASE URL, Load fails
- If no hint is provided, the assembly resolver must probe several location
- Relative search path uses subdirectories of the APPBASE
- Probe path can be augmented using configuration files
- Resultant assembly must match all specified properties (e.g., (policy-adjusted) version, culture)

If the requested assembly cannot be found in the GAC, the assembly resolver then tries to use a CODEBASE hint to access the assembly. A CODEBASE hint simply maps an assembly name to a file name or URL where the assembly file is located. Like version policies, CODEBASE hints are located in both application and machine-wide configuration files. Figure 2.19 shows an example configuration file that contains two CODEBASE hints. The first hint maps version 1.2.3.4 of the `Acme.HealthCare` assembly to the file `C:\acmestuff\Acme.HealthCare.DLL`. The second hint maps version 1.3.0.0 of the same assembly to the file located at `http://www.acme.com/bin/Acme.HealthCare.DLL`. Assuming that a CODEBASE hint is provided, the assembly resolver can simply load the corresponding assembly file and the loading of the assembly proceeds as if the assembly were loaded by an explicit CODEBASE. However, if no CODEBASE hint is provided, the assembly resolver must begin a potentially expensive procedure for finding an assembly file that matches the request.

```
<?xml version="1.0" ?>

<configuration
  xmlns:asm="urn:schemas-microsoft-com:asm.v1"
>
  <runtime>
    <asm:assemblyBinding>
<!-- one dependentAssembly per unique assembly name -->
      <asm:dependentAssembly>
        <asm:assemblyIdentity
          name="Acme.HealthCare"
          publicKeyToken="38218fe715288aac" />
<!-- one codeBase per version -->
        <asm:codeBase
          version="1.2.3.4"
          href="file://C:/acmestuff/Acme.HealthCare.DLL"
        />

        <asm:codeBase
          version="1.3.0.0"
          href="http://www.acme.com/Acme.HealthCare.DLL"/>
      </asm:dependentAssembly>
    </asm:assemblyBinding>
  </runtime>
</configuration>
```

Figure 2.19: Specifying the codebase using configuration files

If the assembly cannot be located using the GAC or a CODEBASE hint, a search is performed through a series of directories relative to the root directory of the application. This search is known as probing. Probing relies on a per-application

relative search path that each application sets using its configuration file. Probing will only search in directories that are at or below the APPBASE directory (recall that the APPBASE directory is the directory that contains the application's configuration file). For example, given the directory hierarchy shown in figure 2.20, only directories `m`, `common`, `shared`, and `q` are eligible for probing. That stated, the assembly resolver will only probe into subdirectories that are listed in the application's configuration file. Figure 2.21 shows a sample configuration file that sets the relative search path to the directories `shared` and `common`. All subdirectories of APPBASE that are not listed in the configuration file will be pruned from the search.

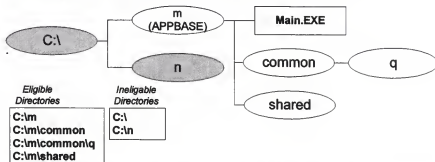


Figure 2.20: APPBASE and the relative search path

```
<?xml version="1.0" ?>

<configuration
  xmlns:asm="urn:schemas-microsoft-com:asm.v1"
>
  <runtime>
    <asm:assemblyBinding>
      <asm:probing privatePath="shared;common" />
    </asm:assemblyBinding>
  </runtime>
</configuration>
```

Figure 2.21: Setting the relative search path

When probing for an assembly, the assembly resolver constructs CODEBASE URLs based on the simple name of the assembly, the relative search path just described, and the requested culture of the assembly (if present in the assembly reference). Figure 2.22 shows an example of the CODEBASE URLs that will be used to resolve an assembly reference with no culture specified. In this example, the simple name of the assembly is `yourcode` and the relative search path is the `shared` and `common` directories. The assembly resolver first

looks for a file named `yourcode.dll` in the `APPBASE` directory. If there is no such file, the resolver then assumes that the assembly is in a directory with the same name and looks for a file with that name under the `yourcode` directory. If the file is still not found, this process is repeated for each of the entries in the relative search path until a file named `yourcode.dll` is found. If the file is found, then probing stops. Otherwise, the probe process is repeated again this time looking for the file named `yourcode.exe`, looking in the same locations as before. Assuming a file is found, the assembly resolver verifies that the file matches all properties of the assembly name specified in the assembly reference and then loads the assembly. If one of the properties of the file's assembly name does not match all of the post-version policy assembly reference's properties, the `Assembly.Load` call fails. Otherwise, the assembly is loaded and ready for use.

Assembly Reference

```
yourcode, Culture=neutral,...
```

APPBASE

```
file://C:/myapp/myapp.exe
```

Application Configuration File

```
<configuration xmlns:asm="...">
  <runtime>
    <asm:assemblyBinding>
      <asm:probing
        privatePath="shared;common" />
      </asm:assemblyBinding>
    </runtime>
  </configuration>
```

Potential CODEBASEs (In order)

```
file://C:/myapp/yourcode.dll
file://C:/myapp/yourcode/yourcode.dll
file://C:/myapp/shared/yourcode.dll
file://C:/myapp/shared/yourcode/yourcode.dll
file://C:/myapp/common/yourcode.dll
file://C:/myapp/common/yourcode.dll

file://C:/myapp/yourcode.exe
file://C:/myapp/yourcode/yourcode.exe
file://C:/myapp/shared/yourcode.exe
file://C:/myapp/shared/yourcode/yourcode.exe
file://C:/myapp/common/yourcode.exe
file://C:/myapp/common/yourcode/yourcode.exe
```

Figure 2.22: Culture-neutral probing

Probing is even more complex when the assembly reference contains a culture identifier. As shown in figure 2.23, the previous algorithm is augmented by looking in subdirectories whose name matches the requested culture. In general, relative search paths should be kept small to avoid excessive load-time delays.

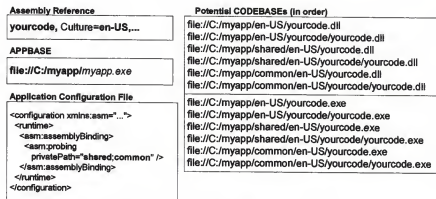


Figure 2.23: Culture-dependent probing

Figure 2.24 shows the entire process the assembly resolver goes through in order to find an appropriate assembly file. Note that once a file is found via whichever mechanism is used, the file must match the version-adjusted assembly reference, otherwise the `Assembly.Load` call fails.

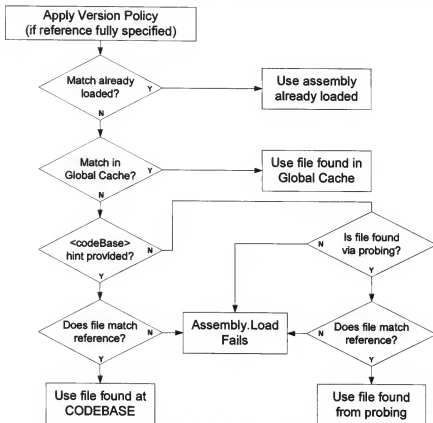


Figure 2.24: Assembly resolution

Versioning hazards

Loading multiple versions of the same assembly is not without risks and problems

- The CLR treats same-named types as distinct when they come from different physical assemblies
- One copy of global/static variables per version
- Types from V2 cannot be passed where types defined in V1 are expected
- Putting globals/static variables in a non-versioned assembly addresses the former
- Avoiding using versioned types as parameters addresses the latter

The discussion of how the assembly resolver determines which version of an assembly to load focused primarily on the mechanism used by the CLR. What was not discussed was what policies a developer should use to determine when, how, and why to version an assembly. Given the ancient nature of the platform, it is somewhat difficult to list a set of "best practices" that are known to be valid based on hard-won experiences. However, it is reasonable to look at the known state of the CLR and extrapolate a set of guidelines.

It is important to note that assemblies are versioned as a unit. Trying to replace a subset of the files in an assembly without changing the version number will certainly lead to unpredictability. To that end, the remainder of this section looks at versioning with respect to an assembly as a whole rather than versioning individual files in an assembly.

The question of when to change version numbers is an interesting one. Obviously, if the public signature of a type changes, the type's assembly must be given a new version number. Otherwise, programs that depend on one version of the type signature will get runtime errors when a type with a different signature is loaded. This means that if you add or remove a public or protected member to a public type, you must change the version number of the type's assembly. If you change the signature of a public or protected member of a public type (e.g., adding a method parameter, changing a field's type), you also need a new assembly version number. These are absolute rules. Violating them will result in unpredictability.

The more difficult question to answer relates to modifications that do not impact the public signature of the assembly's types. For example, changes to a member that is marked as `private` or `internal` are considered non-breaking changes, at least as far as signature matching is concerned. Since no code outside of your assembly can rely upon `private` or `internal` members, having signature mismatches occur at runtime is a non-issue. Unfortunately, signature mismatches are only the tip of the iceberg.

There is a reasonable argument to be made for changing the version number for every build of an assembly, even if no publicly visible signatures have changed. This approach is supported by the fact that even a seemingly innocuous change to a single method body may have a subtle but very real rippling effect on the behavior of programs that use the assembly. By giving each build of an assembly a unique version number, code that is tested against a particular build won't be surprised at deployment-time.

The argument against giving each build of an assembly a unique version number is that "safe" fixes to the code won't be picked up by programs that are not rebuilt against the new version. This argument doesn't hold water in the face of publisher policy files. Developers who use unique version numbers for every build are expected to provide publisher policy files that state which versions of their assembly are backwards-compatible. By default, this gives consumers of the down-level assembly version an automatic upgrade to the

newer (and hopefully faster or less buggy) assembly. For times when the assembly's developer guesses wrong, each application can use the `publisherPolicy` element in their configuration file to disable the automatic upgrade, in essence running the application in "safe-mode."

As discussed in great detail earlier, the CLR assembly resolver supports side-by-side installation of multiple versions of an assembly via codebase hints, private probe paths, and the GAC. This allows several versions of a given assembly to peacefully co-exist in the file system. However, things become somewhat unpredictable if more than one of these assemblies is actually loaded into memory at any one time, either by independent programs or by a single program. Side-by-side execution is much harder to deal with than side-by-side installation.

The metadata for an assembly has a distinguished attribute that allows the developer to specify whether multiple versions of the assembly can be loaded at the same time. This metadata attribute has four possible values. An assembly can be marked as safe for side-by-side execution in all scenarios. An assembly can be marked as safe for side-by-side execution provided only one version of the assembly is loaded per-process. An assembly can be marked as safe for side-by-side execution provided only one version of the assembly is loaded per-AppDomain. Or finally, an assembly can be marked as unsafe for side-by-side execution in any scenario, in essence indicating that only one version of the assembly can be in memory for the entire machine. At the time of this writing, this metadata bit is ignored by the assembly resolver and loader, however, it does act as a hint that hopefully will be enforced in future versions of the CLR.

Loading multiple versions of the same assembly is not without risks or problems. The primary problem of supporting multiple versions in memory at once is that to the runtime, the types contained in those assemblies are distinct. That is, if an assembly contains a type called `Customer`, when 2 different versions of the assembly are loaded, there are two distinct types in memory each with their own unique identity. This has several serious downsides. For one, each type has its own copy of any static fields. If the type needed to keep track of some shared state independent of how many versions of the type had been loaded, it could not use the obvious solutions of using a static field. Rather, the code would need to be written with versioning in mind and store the shared state in a location that is not version-sensitive. One approach would be to store the shared state in some runtime-provided place such as the ASP.NET application object. Another approach would be to define a separate type that only contained the shared state as static fields. This type could be deployed in a separate assembly that would never be versioned, thus ensuring only one copy of the static fields would be in memory for a given application.

Another problem related to side-by-side execution arises when versioned types are passed as method parameters. If the caller and callee of a method

have differing views on which version of an assembly will be loaded, the caller will pass a parameter whose type is unknown to the callee. This can be avoided by always defining parameter types using version-invariant types for all public (cross-assembly) methods. More importantly, these shared types need to be deployed in a separate assembly that itself will not be versioned.

Summary

- The assembly is the "component" of the CLR
- All types belong to exactly one assembly
- Assemblies carry a digital signature from the developer who created them
- Assemblies support multiple versions loaded simultaneously in the same program
- Assemblies can be automatically downloaded from the Internet

Module 3

Code Access Security

Traditional security on the Windows platform has been process-centric. All code within a process runs at the same level of privilege (impersonation aside). This worked fine until the component revolution. The CLR layers new protections on top of the native security provided by the operating system. This new security model is called Code Access Security (CAS), and it allows assemblies to run with different permissions within the same physical process. This is good news for end users, but it also poses yet another call to developers to pay attention to security in their applications. This module covers the CAS permissions, policy, and the stack walking mechanism.

After completing this module, you should be able to:

- understand why code access security is important
- describe the three places where security policy comes from
- be aware of the different forms of evidence used to make policy decisions
- use the CASPOL utility to learn about permissions on an assembly
- describe what happens when a permission is demanded
- understand why Assert exists and when it should be used

Code Access Security: Motivation

The traditional process-centric security model breaks down in a component-oriented world. Code Access Security addresses this issue by layering a component-aware security model over any existing operating system security model

History

Historically, security has been process-centric

- Each process is assigned a token by the system
- Token contains user identity and group identifiers
- When one process creates another, the token is duplicated to the new process
- System worked reasonably well since one vendor generally shipped all the code for an application

permissions
auditing

When Windows NT 3.1 shipped in June 1993, most programs being written for it were generally monolithic applications that sat on top of system-provided DLLs. One might find an occasional custom DLL in one of these old applications, but generally that DLL would be provided by the same vendor that built the EXE, simply because some Win32 functionality requires packaging code in DLLs. Figure 3.1 shows a typical process during this time period.

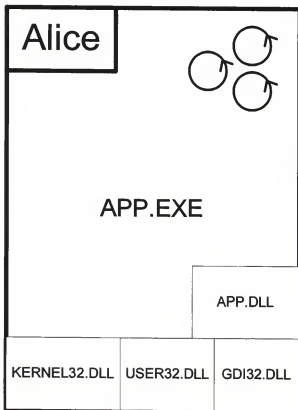
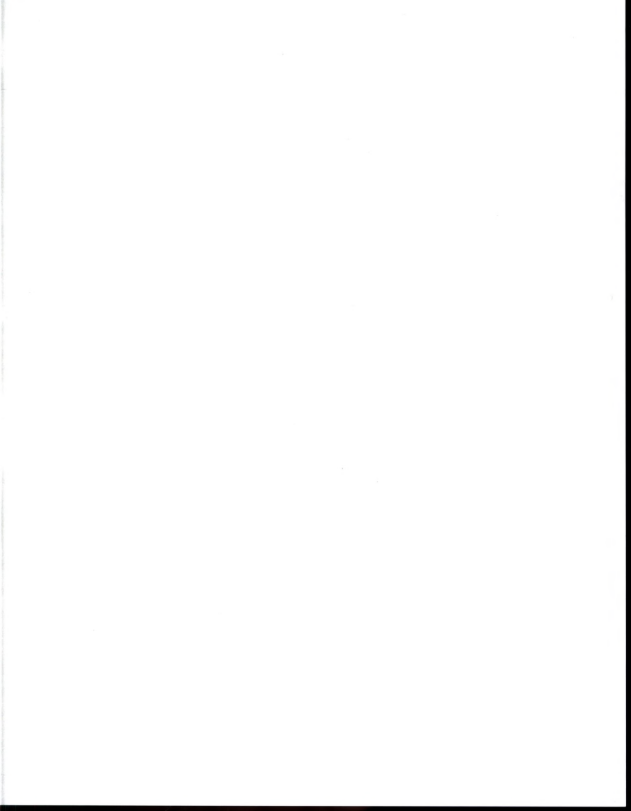


Figure 3.1: Monolithic applications

Windows NT 3.1 was therefore designed with a traditional process-oriented security model. Each process is assigned a security access token (and no process may run without one). This token holds the results of authentication: it tells the system the identity of a user, the groups that user is a member of, etc. The idea being that whenever any threads in the process attempt to acquire resources like files, registry keys or other kernel objects, the system could simply look at the token associated with that thread's process to

determine on whose behalf the thread is operating. Access checks and auditing could then proceed using the token as a source of user information. When a thread in a process creates another process, that new process gets a copy of the old process's token, which makes sense.

So what about security? If an application did something bad, it wasn't too hard to figure out which vendor was responsible. Also, the lack of widespread Internet connectivity meant that the only threat to a typical consumer was the virus. It was a pretty safe and mundane world. This model worked really well until the Internet became mainstream. Around the same time some smart guy somewhere decided that it'd be a good idea to start gluing processes together from random DLLs purchased from random vendors using "component technology".



The Component Revolution

Dynamically loaded components (ala COM) break a process-centric model

- The COM revolution led to building processes from DLLs
- DLLs now built by multiple vendors
- Each DLL runs at the same level of privilege (process token)
- To sandbox a DLL, it must be run in a separate process, which is expensive and painful to administer

When COM really took off in 1995, Windows security was still process-centric. But now people started buying components from third party vendors and gluing applications together from COM components, which were always packaged in DLLs for performance (making cross-process calls is much more expensive in COM). No longer could a user tell where her application really came from - it ultimately came from multiple vendors. Figure 3.2 shows the new state of affairs. At the same time, with more and more systems being attached to the Internet, new avenues of attack were being opened up.

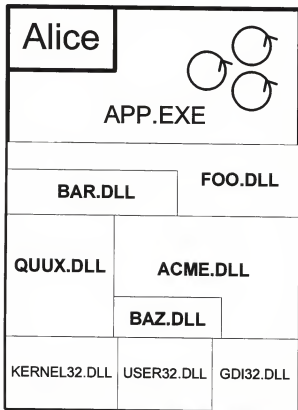


Figure 3.2: Component-based applications

The crux of the problem is that all the DLLs in a process run at the same level of privilege. When a DLL is loaded, its code is simply mapped into the process address space; the security system cannot tell it apart from the code in the host EXE. This becomes particularly frightening considering that most users run Windows with administrative privileges.

The Internet phenomenon and ActiveX controls compounded the problem. Now consumers, using their browsers, could download component DLLs over the Internet to "enhance their browsing experience". Any site could ship a DLL to a client, good sites and bad sites.



Authenticode

Authenticode was a smokescreen

- Attempted to make component vendors accountable
- Punitive solution, not a preventative one
- By the time you're in court, damage has already been done
- Even well-meaning components often have gaping security holes due to weak implementations

In an attempt to give consumers some protection, Microsoft introduced Authenticode. This technology allows component vendors to digitally sign their component DLLs so that consumers can verify the origin of the code they download and make a decision whether or not to trust it. If a consumer is willing to install shrink-wrapped software on their system from a particular vendor, then why not also install downloaded DLLs from the same vendor?

Unfortunately Authenticode didn't go far enough. It only addresses accountability: you can bring suit against the vendor of a bad piece of code because now you know who authored it, the code has his nonrepudiable signature on it. Authenticode is punitive, not preventative. By the time you're in court, the damage has already been done.

The other problem that Authenticode didn't address is that most vendors don't write bad code on purpose. Most security breaches are due to attackers exploiting well-meaning but faulty implementations in code. Take figure 3.3 as an example. What's wrong with that code? Assume for a moment that code similar to this is running in your web server in an ISAPI application, at low isolation (for performance).

```
void main() {  
    foo();  
}  
  
void foo() {  
    char buf[1024]; // surely this will be big enough!  
    readUserNameFromForm(buf);  
    logUserName(buf);  
}
```

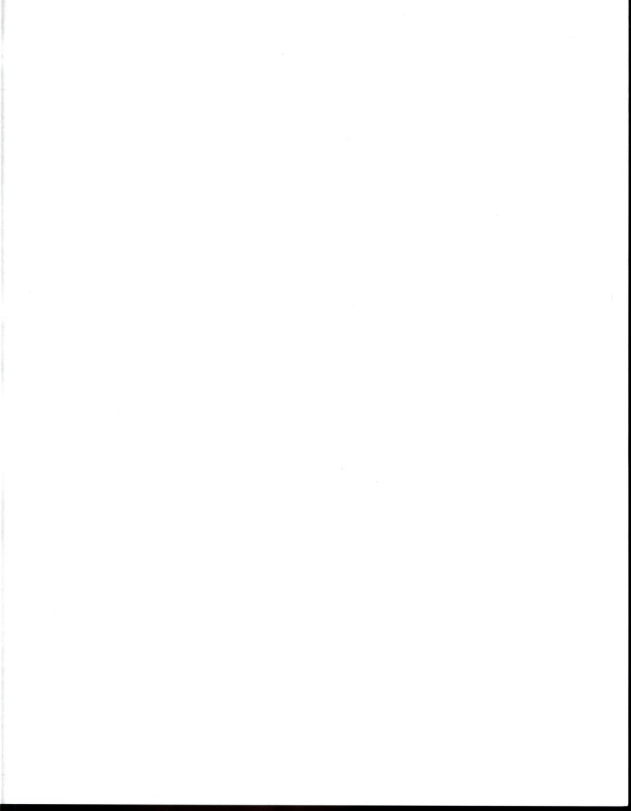
Figure 3.3: Well-meaning, but ultimately bad code

The code has what is likely the worst possible security-related bug, a stack buffer overflow. An attacker will simply send longer and longer strings in their HTTP request to your form, until he eventually overflows the stack. What's stored on the stack right next to the buffer? The return address from the function `foo()`. Once he overflows this, when `foo()` returns it won't return to `main()`, but rather to whatever address he sent in his string. Ultimately the attacker will send a small program (x86 instructions) instead of a name, and overflow the buffer such that the return address points into the stack buffer, and when `foo()` returns, the program will start executing the attacker's code.

This situation looks pretty bleak, but it gets worse. What security context is the attacker's code executing in? Well, since the ISAPI application is running in-process in the web server, it's running in `INETINFO.EXE`, which runs in the security context of `SYSTEM`. That's basically the equivalent of root in Unix. The typical program that would be uploaded is about a six-line program that uses `Wininet` to download a larger, more sophisticated program of the attacker's choosing onto the victim's machine, calling `CreateProcess` on that program (the

new process will also run as SYSTEM as I mentioned earlier), and then the web server will be terminated quietly with a call to `ExitProcess` from the original attacker's program. That way it'll take the administrator awhile to notice that anything is wrong, while the new program starts attacking the password database on the web server in an attempt to penetrate other computers on the victim's network.

The whole problem is that in a process-centric security system, in order to isolate code at a lower level of security, it must be removed and placed in a separate process running with a different identity. This reduces performance due to cross-process communication, and increases administrative burden, as the administrator must now manage not only accounts for real users, but accounts for daemon processes as well. It's generally not done and the problem is simply ignored. Note that the web server problem was solved in IIS 4 by introducing the notion of an isolated web application, but there's a performance hit here as you'd expect, and many ISPs happily run your ASP applications in-process.



Code Access Security

CAS layers component-centric security over the security model of the OS

- CAS provides a preventative, rather than a punitive model
- Verifiable code isn't subject to buffer overflow attacks
- Policy-driven assignment of permissions at assembly load time enables component-aware prevention of malicious activities
- CLR may grant access to resource, but underlying OS still has a say in the matter

The CLR provides a much more comprehensive solution to this problem by layering a component aware security model on top of what the underlying operating system provides. Note that because of the layering, the original native security system in the OS is still in place; this system simply allows you to place additional restrictions on individual components, which is exactly what we need. Figure 3.4 shows this layering.

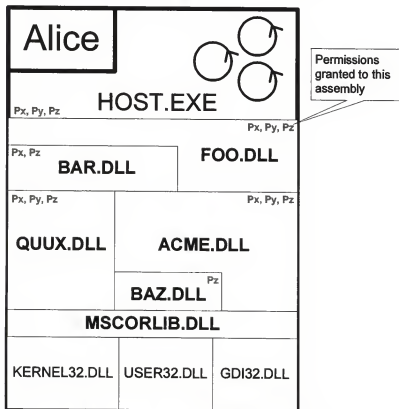


Figure 3.4: CAS is layered on top of OS native security

CAS is preventative. It prevents components from doing things they aren't allowed to do, without forcing them to run in separate processes. Administrators can grant permissions to components based on where the component came from (Internet, Intranet, local computer, etc.) or who authored the component. The latter case is very much like Authenticode, except CAS deals with access control rather than simply accountability.

Why did it take so long to get a preventative, rather than a punitive solution? Implementing CAS is only feasible in a world where components can't cheat. There are no buffer overflow attacks in verifiable code, so even bad code should theoretically not cause problems (although with implementation experience we may discover new attacks that didn't exist before; it's doubtful that the CLR is perfect). Arguably the advance in component security is one of the most compelling reasons to move away from the old world and toward the CLR.

Security Policy

Assemblies are granted permissions based on their origin. Code Access Security (CAS) Policy dictates the permissions granted to any assembly.

Evidence

The loader discovers evidence of the origin of code

- Evidence describes the code being loaded
- Used to determine the permissions granted to an assembly
- Evidence takes many forms
- Zone, URL, X.509 certificate, strong name, etc.
- Custom evidence also possible

You may be wondering how the system figures out just what permissions are granted to any given component. First of all, permissions are granted to assemblies, not to individual classes. All classes in an assembly are granted the same basic set of permissions. When an assembly is loaded, the loader discovers Evidence about that assembly - where it came from, who authored it, etc. This evidence is passed to policy to decide which permissions to grant the assembly.

Evidence can take many forms. Figure 3.5 lists the built-in classes of evidence that exist today; you can add your own if you like. Given the different classes of evidence, you can see the choices an administrator has when figuring out how to set up policy. He can grant permissions on the basis of the Zone an assembly is loaded from (Internet, Intranet, MyComputer, etc.), the specific URL that the assembly came from, the assembly publisher's identity (publishers are identified by X.509 certificates, just like in Authenticode), the assembly's strong name, etc.



Zone
Site
Url
Publisher
StrongName

Figure 3.5: Some built-in classes of evidence

mscorlib.exe - config permissions
mscorlib.msc

Permissions

Permissions express details about sensitive operations

- Permission objects are used for two things: grants and requests
- `IPermission` expresses this, and `IsSubsetOf` allows comparison between requests and grants
- Permission objects are serializable to allow storage in policy files
- `PermissionSet` allows grouping of different classes of permissions
- Several built-in permission classes already exist
- Possible to extend with your own classes

So what sorts of permissions exist? Figure 3.6 shows a list of the built-in code access security permission classes, and gives you an idea of what types of operations are considered sensitive in the CLR. Note that it's possible to extend this set of classes with your own application-specific permissions as well.

```
DBDataPermission
PrintingPermission
MessageQueuePermission
DnsPermission
SocketPermission
WebPermission
EnvironmentPermission
FileDialogPermission
FileIOPermission
IsolatedStoragePermission
ReflectionPermission
RegistryPermission
SecurityPermission
UIPermission
```

Figure 3.6: Some built-in code access security permission classes

Let's code up an example to help make all this concrete. Figure 3.7 shows the file IO permission request described earlier. Note the pattern. To ask the system whether or not a permission is granted, you create a permission object describing the specific details of what you want to do, and then you call `Demand()`. If the permission is denied, the call to `Demand` will result in a `SecurityException` being thrown. To decide whether permission is granted or denied, the runtime compares the request with the caller's permissions using `IsSubsetOf` (there are more details to come, but this description suffices for now).

```
using System.Security.Permissions;

class Test {
    public void readSomeFile() {

        // describe your request
        IPPermission request = new FileIOPermission(
            FileIOPermissionAccess.Read,
            @"c:\foo\bar\quux.txt");

        // make the request
        request.Demand(); // throws SecurityException on
failure

        // if we're still here, the demand succeeded!
    }
}
```

Figure 3.7: Example: requesting a permission



Policy Levels

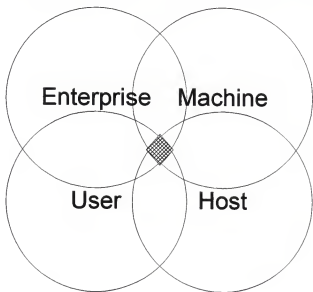
On a given machine policy is arranged in four levels

- Enterprise policy file, administered by the domain admin
- Machine policy file, administered by the sysadmin
- User policy files, administered by individual users
- Host policies, provided at runtime by the app hosting the CLR
- Resulting permission set is intersection of all four levels

In order to provide both administrators and individual users the ability to protect their systems, there isn't simply one policy file that maps evidence to a set of granted permissions. Instead, there are three separate files, one governing policy for the network enterprise, one for the machine as a whole (both of which stored in a central directory on the machine), and one for each user of that machine (stored in the user's profile directory). These are the persistent policy levels, and fundamentally consist of serialized evidence to permission mappings. There is also a fourth transient policy level--transient because it's not necessarily stored in a file somewhere--rather it's provided dynamically by the host of the CLR. Only enterprise or domain administrators can make changes to enterprise policy - redistributing the new policy file to each computer in the enterprise (using a Microsoft Installer (MSI) file, Windows 2000 group policy, or Systems Management Server (SMS) 2.0). Similarly, only a system (machine) administrator can make a change to the machine-wide security policy for a given machine.

When an assembly is loaded, the four policy levels are applied in a well-determined order: enterprise, machine, user, and host. The idea behind this is to allow the enterprise-wide and machine-wide settings to take precedence over all others, so that a sysadmin can provide an enterprise-wide or system-wide security policy that cannot be compromised by individual user policies. Ultimately, you can think of each policy level as a transformation, with the evidence as input, and a permission set as output. The set of allowed permissions is the intersection of the results from evaluating all four levels, which means (for example) a user can only alter the results of the machine policy level by restricting permissions - individual users cannot open holes in the system that a sysadmin doesn't want opened up.

Figure 3.8 shows the four policy levels and how we take the intersection of the permissions from each to determine the set of allowed permissions.



Intersection of results form
the set of allowed permissions

Figure 3.8: Policy levels

*enterprise security.config - sec. for ent
.... \config is where all config files
are.*



Code Groups

Each policy level consists of a tree of code groups

- A code group maps a membership condition to a set of permissions to grant
- Code groups are arranged in a tree
- A child code group will only be evaluated if the parent's criteria was satisfied
- Allows non-programmers to combine criteria using boolean logic

A given policy level consists of a tree of objects called Code Groups. This is a somewhat confusing name for a really simple idea - a single criteria that maps code (assemblies) to a permission set. The criteria is a test against the evidence. For instance, you might have a code group that says "if an assembly was published by ACME corporation, it should be granted the following permissions...". Figure 3.9 shows the basic structure.

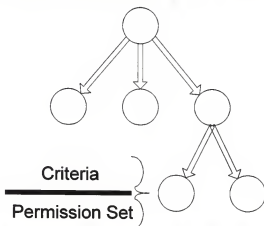


Figure 3.9: A policy level consists of a tree of code groups

So why are code groups (criteria to permission set mappings) arranged in a tree? The idea is to allow non-programmers to be able to produce simple boolean-like policies; children of a given code group are not evaluated unless the parent code group's criteria is satisfied. So an administrator could set up a code group like the one in figure 3.10 to basically say "grant these permissions only if both criteria are met". Figure 3.11 shows another example that says "grant these permissions if either criteria is met".

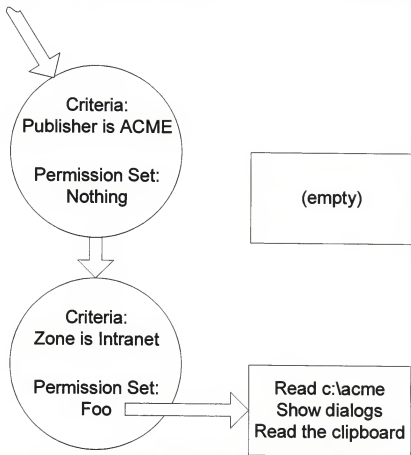


Figure 3.10: Modeling a boolean AND with two code groups

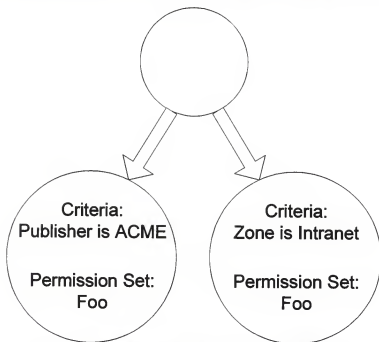


Figure 3.11: Modeling a boolean OR with two code groups

It's interesting to note that code groups refer to permission sets by name, so they can be shared. This isn't a big deal, but it allows expressions such as the OR expression shown in figure 3.11 to be implemented without having to duplicate permissions. So there's technically another part to a policy level besides the tree of code groups: a list of the named permission sets referred to by the code groups. So a code group technically consists of a criteria and a reference to a named permission set (for the lawyers in the house).

Declarative Permission Requests

Each assembly can make declarative permission requests

- Assemblies can effectively filter permissions allowed by policy
- Allows an assembly to state the minimum permission set it does need
- Allows an assembly to refuse permissions it doesn't need
- Assembly-level attributes control these requests
- A `PermissionSetAttribute` that refers to a file is a good way to go

The story so far has the loader gathering evidence, submitting that evidence to the criteria in the three different policy levels, and intersecting the resulting permissions from those levels to form the set of allowed permissions. Now the only step left is to take the allowed permission set (the permissions allowed by policy) and determine what the true granted permission set should be (and you thought we were done!) Really all this last step does is allows the assembly itself to have some say in what permissions are assigned to it.

There are three declarations an assembly can make (using declarative attributes): minimum, optional, and refused permission sets. The minimum permission set simply says that if the assembly isn't allowed this minimal set of permissions, it shouldn't be allowed to run at all (perhaps the assembly wouldn't be able to get any work done if it couldn't read files from the local file system, say).

The other (optional and refused) permission sets are really just two different ways of saying the same thing: the assembly doesn't want all possible permissions allowed by policy. Why would you want to reduce the permissions granted to an assembly that you've authored? Well, this gives your assembly a way to further protect itself against misuse. Remember that even the best programmers sometimes make mistakes that can be exploited as security holes. Even though verifiable code eliminates holes like stack buffer overflows (one of the most commonly exploited bugs in C and C++ applications), and the CAS stack walking mechanism is designed to prevent less privileged code from using your assembly to do bad things, you can never be too careful. Administrators will invariably make mistakes in their policy configuration, and by refusing permissions that you know your assembly will never need, you may just save yourself from being misused.

As mentioned earlier are two declarations you can make that limit the permissions your assembly will be granted by the CLR. The first is to directly refuse the permissions you know you will never need. The drawback to using this approach is that new permissions may be introduced (remember the system is extensible) that you weren't aware of when you authored your assembly. So another approach exists as well, instead of refusing permissions, simply list the permissions you know you might need, and let the runtime deny you all others implicitly. This is done by specifying the "optional" permission set for your assembly.

Here's the algorithm used by the CLR to compute the granted permission set, given the allowed permission set from policy, and the permission requests made by the assembly:

1. If none of the three assembly level attributes exist (`RequestMinimum`, `RequestOptional`, or `RequestRefuse`), simply grant all permissions that were allowed by policy. End of story.

2. Establish an empty set of permissions, let's call this the granted permission set. When we're done, this will be the set of permissions assigned to the assembly.

3. If a RequestMinimum attribute is present, determine whether those permissions were in fact allowed by policy. If they were not allowed, throw a PolicyException and do not load the assembly at all (and quit this algorithm). If they were allowed, add those permissions into the granted permission set.

4. If a RequestOptional attribute is present, take the intersection of the allowed permission set and the optional permission set and add that into the granted permission set. This sounds kind of complicated, but all that's really happening here is that we're filtering out any permissions that were allowed by policy, but that you didn't mention in your optional request.

5. If a RequestRefuse attribute is present, subtract that permission set from the granted permission set.

Figure 3.12 shows some code that refuses file I/O permissions.

```
using System;
using System.IO;
using System.Security.Permissions;

[assembly: FileIOPermission(
    SecurityAction.RequestRefuse,
    Unrestricted=true)]

class ep {
    static void Main() {
        Console.WriteLine("Contents of foo.txt:");
        StreamReader r = File.OpenText("foo.txt");
        Console.WriteLine(r.ReadToEnd());
    }
}
```

Figure 3.12: Annotating an assembly with a permission refusal

Figure 3.13 shows some code that requests minimum and optional permissions for an assembly. Note that because you'll usually want to specify multiple permissions, it's a good idea to create a permission set programmatically and serialize it to an XML file. You can then simply refer to that XML file via the PermissionSet attribute. Figure 3.14 shows the technique used to create one of these files.

```

using System;
using System.IO;
using System.Security.Permissions;

[assembly: PermissionSetAttribute(
    SecurityAction.RequestMinimum,
    File="min_perm.xml")]
[assembly: PermissionSetAttribute(
    SecurityAction.RequestOptional,
    File="opt_perm.xml")]

class ep {
    static void Main() {
        Console.WriteLine("Contents of foo.txt:");
        StreamReader r = File.OpenText("foo.txt");
        Console.WriteLine(r.ReadToEnd());
    }
}

```

Figure 3.13: Annotating an assembly with permission requests

```

// create the permissions you need
FileIOPermission p1 = new FileIOPermission(
    PermissionState.Unrestricted);
RegistryPermission p2 = new RegistryPermission(
    PermissionState.Unrestricted);

// group them into a single permission set
PermissionSet pset = new PermissionSet(
    PermissionState.None);
pset.AddPermission(p1);
pset.AddPermission(p2);

// convert into XML
SecurityElement miniDOM = pset.ToXml();

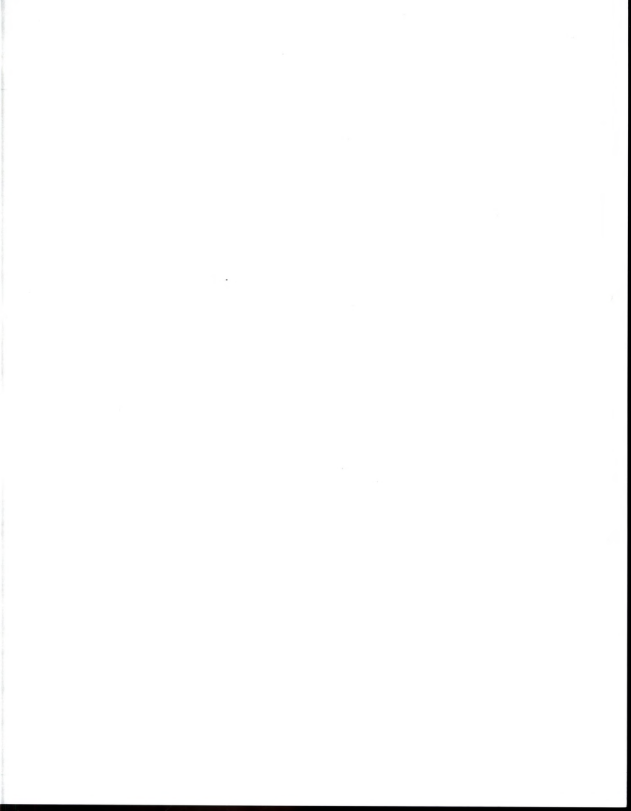
// put it somewhere (file IO omitted for brevity)
Console.WriteLine(miniDOM.ToString());

```

Figure 3.14: Creating a serialized permission set

One final note - be careful about refusing permissions that extensibility points in your assembly might require. For instance, if you fire events or call other's code via interfaces or delegates, you will effectively be restricting what those event handlers and interface/delegate implementations can do. Ignoring some technicalities, the CAS stack walking mechanism says that when you call a

piece of code that code can't exercise any permissions that you yourself were not granted. Just be aware that when you refuse permissions, you're effectively making security policy for your own assembly and for other assemblies that you might call, and it's arguably the administrator's job to make security policy, not the assembly author's job. It remains to be seen whether or not allowing assemblies to perform these requests was a good or bad idea, but it helps to understand the basic tensions involved.



Administering Policy

MSCORCFG.MSC is used to administer (among other things) enterprise, machine and user policy levels

- Can be used to read and modify policy
- Can be used to check the permissions assigned to an assembly
- CASPOL.EXE provides command-line driven management (albeit more difficult to use)

Administrators use the CASPOL tool to modify both machine and user policy files. CASPOL can be used to review the current policy, change policy, turn code access security on or off entirely, etc.

CASPOL is a very low-level tool that requires the administrator to either author XML by hand, or to write code to serialize permission sets to XML. It's not pretty. MSCORCFG.MSC (as shown in Figure 3.15) is an MMC console-based tool that provides a graphical view of the hierarchy of code groups in the enterprise, machine and user policy levels.

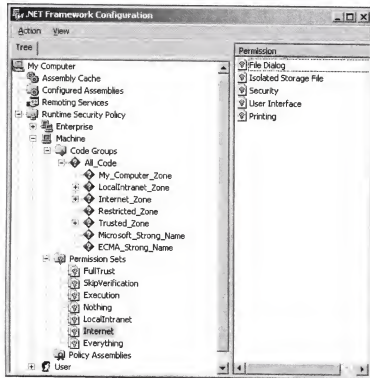


Figure 3.15: MSCORCFG.MSC MMC Snap-In

Some simple experiments you can run with CASPOL are to test what permissions an assembly will have based on policy, or to see which code groups it matches. For instance, given a single assembly `foo.dll` that's both located on your local machine and on the Internet somewhere, the following command lines will produce very different results (assuming you've still got the default policy that ships with the CLR):

```
caspol -resolveperm c:\foo.dll
```

```
caspol -resolveperm http://www.acme.com/foo.dll
```

The first line will grant virtually unrestricted access to foo.dll, because the code is installed locally (local code is very highly trusted in the current default policy). The second line will produce a much more restricted set of permissions.

The CAS Stackwalk

Programmers often use layering to simplify code, but this often opens up security holes as misuse of one layer can lead to the compromise of another. Code Access Security guards against these attacks via a stack walking mechanism.

The Luring Attack

Attackers may try to lure trusted components into doing evil things

- Evil components generally want to elevate their privilege level
- Privilege escalation can occur by luring a more privileged component
- One policy is that of isolation: don't let the components talk to each other at all
- CAS takes a more benevolent approach by monitoring the conversation
- The IStackWalk interface is the key abstraction here

On a given computer system, an attacker that has a restricted level of privilege will attempt to elevate that privilege. Usually an attacker wants to achieve administrative control over a computer. Many attacks are indirect attacks against components that seem to be safe, but in actuality perform operations that the attacker can leverage.

These indirect attacks are often called "luring" attacks, because the attacker lures a component running in some more trusted security context to do something on his behalf. Here's an example of such an attack. An evil process, sandboxed off in a logon session for a very low privileged user, wants to increase its level of privilege. So it occasionally checks to see if a copy of EXPLORER.EXE is running, which generally indicates that an interactive user is present. Say the network administrator logs into this box, and our evil process detects this, by noticing that EXPLORER.EXE is now running. The evil process calls FindWindow to discover the window handle of the shell, then begins posting messages to that window (WM_KEYDOWN and friends) to simulate the pressing of keys. You can try this experiment at home: press 'Control-Escape', 'r', and look at what you end up with - a dialog box asking for the name of a program to run. The evil process can continue sending keystrokes to place the full path to his executable in that dialog, followed by 'Enter' to submit it to the shell.

At this point, the shell calls CreateProcess (indirectly). CreateProcess runs the new process in the same security context as the process who called CreateProcess (in this case EXPLORER.EXE), which means the attacker now has a copy of his program running under in the security context of the interactive user, who also happens to be the network administrator. Now our evil program can happily reconfigure every computer on the network to his delight.

It turns out that this luring attack isn't really possible (except on Windows 2000 prior to Service Pack 1, where a nasty little bug allows a variation of this attack). Processes in non-interactive logon sessions are segmented into private window stations, and window stations completely isolate windows from outside interference (you can't send messages across window station boundaries). But this policy of isolation means erecting a wall between components that cannot be penetrated, even by good, well-meaning components that need to communicate.

What we need is protection against luring attacks without resorting to a policy of total isolation. The stack walking mechanism provided by code access security in the CLR addresses this in an elegant way. The interface IStackWalk (shown in figure 3.16, implemented by all code access permission objects, is key to making this work.

```
namespace System.Security {  
    public interface IStackWalk {  
        void Assert();  
        void Demand();  
        void Deny();  
        void PermitOnly();  
    }  
}
```

Figure 3.16: The IStackWalk interface



Demand

A permission demand percolates up the call stack

- Runtime verifies that each assembly was previously granted the permission being demanded
- Security exception results if all callers were not granted the permission
- Callers may optionally place markers (deny/assert) on the stack that short-circuit subsequent stack walks initiated by lower-level components

Most things you do in the CLR aren't security sensitive operations. When you're adding two numbers, concatenating strings, etc., the classes you use to do this (`Int32` and `String`, perhaps) aren't going to be checking to see if you have permission to do that; anyone can do these things. However, there are lots of classes that do perform security sensitive operations. `FileStream` is a great example. When you construct a `FileStream` object, you specify a filename and what sort of access you need, and the `FileStream` constructor demands that you have been granted permission to open that file (via policy).

The `FileStream` object does this by creating a `FilePermission` object and populating it with the details of the access request - which file, and what level of access (read, write, etc.) was requested. It then calls `Demand()` on that permission. This key method initiates a stack walk, checking the permissions of all callers on the call stack, beginning with the direct caller of the method that invoked `Demand()`. Each assembly participating in that call stack has already been granted a set of permissions via policy, and this stack walk is simply the mechanism used to check those permissions.

With `Demand`, if at any point in the stack walk we find code that is not granted all the permissions being demanded, a `SecurityException` is thrown and we're done. An attacker trying to lure a trusted component will be caught by this stack walk if he asks that component to exercise a permission that his assembly has not been granted.

To reiterate, when a `Demand` is issued, all callers in the call stack being searched must have all permissions being demanded. Otherwise a `SecurityException` will be thrown.

Deny and PermitOnly

Each stack frame can restrict its effective permission set

- Before making a call, you can restrict your permission set
- Deny is used when you know specific permissions you want to remove
- PermitOnly is used when you know the set of permissions you want to allow
- Generally you should avoid making these sorts of policy decisions

Many classes are built with extensibility hooks such as callback interfaces, events, etc. Often even more dynamic mechanisms such as reflection can be used to call into code. If you're writing a component that is likely to be granted a high level of privilege, it's possible to throttle those privileges before making these calls to other, potentially unknown code. The way this is done is by placing a Deny or PermitOnly permission set on your stack frame. To do this, create a permission (or a permission set) and call Deny to effectively subtract those permissions, or call PermitOnly to directly restrict the permissions for your stack frame. Now when you make calls to other objects, if a CAS stack walk reaches you, demanding one of the permissions you excluded, that demand will fail. It's really quite straightforward, and figure 3.17 shows an example.

```
public void DoSomeWorkThenMakeCallback(CallbackDelegate d)
{
    // do some work...

    // restrict our permission set before making the call
    FileIOPermission p = new FileIOPermission(
        PermissionState.Unrestricted);
    p.Deny();

    // make the call
    d.HereIsTheCallback(); // won't be able to open files

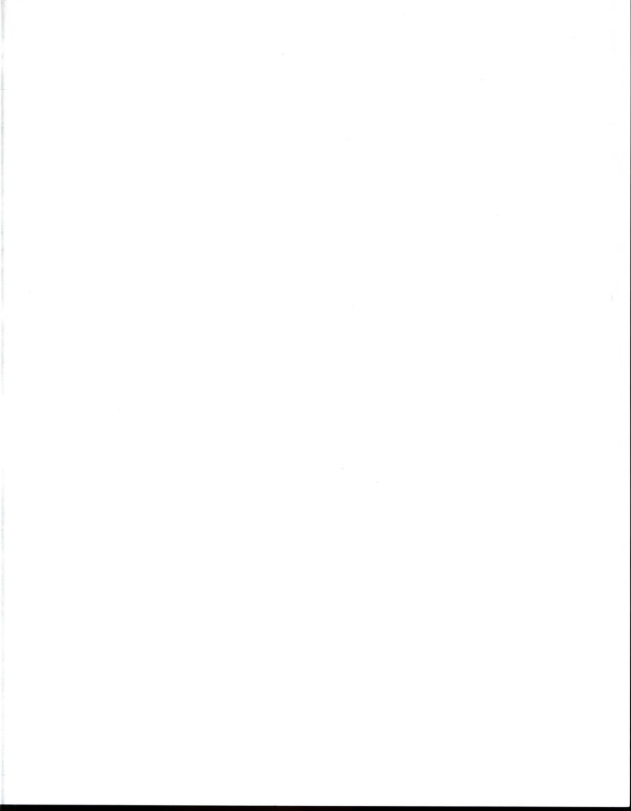
    // returning removes our stack frame (and the Deny)
}
```

Figure 3.17: Example: denying permissions before making a call

So should you use this all over the place to protect yourself? Before you fire an event, make a call through a callback interface, etc., should you think about all the different permissions you should deny? No. Remember that the assembly you're calling into also participates in the stack walk when it asks system components to do sensitive work. Let system policy take care of this for you. If you call to a component that was downloaded from the Internet, it won't have nearly the level of privilege as a locally installed component. You don't want to be in the business of making policy.

So if you shouldn't make policy, then why do these functions exist? Well, for one thing, some components are specifically written to provide more dynamic security policy decisions (general purpose components don't do this). Also, there may be special cases, such as calling into dynamically generated code, where you need to sandbox the code regardless of its policy-assigned permissions. The point is that while you generally will not need these functions, you'll be relieved to find that they exist when you do need them.

If you do use these functions, be aware that there is only one Deny permission set. There is only one PermitOnly permission set, etc. In other words, if you want to deny two permissions, don't call Deny twice on two separate permission objects, because the second Deny will simply overwrite the first (it won't form the union as you might expect). To use multiple permissions, either form a PermissionSet or use Union/Intersect to form a more sophisticated set of permissions. The key is that each time you call Deny, the set of permissions you're denying replaces the set that used to be in effect.



Assert

Trusted components may need to specialize generic demands

- You can't make an omelet without breaking some eggs
- Assert allows a trusted component to do work that its caller isn't allowed to do
- Trusted component demands a more specific permission before asserting its own authority
- Example: controlled calls to unmanaged code are a fact of life
- Assertion ultimately used to allow a more fine-grained policy

As the saying goes, in order to make an omelet you have to break some eggs. Consider the position of the poor person at Microsoft who was tasked with implementing `FileStream`. In order to open a file, she needed to call the unmanaged function `CreateFile`. (Remember, the CLR is not an operating system; to get work done the CLR often needs to make calls to the native operating system underneath.) So she makes this call. What happens?

The interop layer in the CLR is one of the most security sensitive beasts you'll encounter. A call leaving managed space and entering the unmanaged world is a dangerous call indeed. Who is to say what will happen when we leave the confines of the CLR? Out there in the unmanaged world, code can use pointers to search through our managed heap for passwords, or corrupt our stack frame. Even well-meaning code written with the best of intentions can have bugs that allow a bad guy to perform these sorts of attacks. So the interop layer will always demand a very special permission before making the native call. That permission is a `SecurityPermission` named `UnmanagedCode`. To make a call to native code requires this permission.

Now it just so happens that the assembly hosting `FileStream` and the other built-in classes that Microsoft ships with the CLR is trusted by policy to make calls to unmanaged code. But what about the assembly that you're writing that needs to use `FileStream` to get work done? When that stack walk comes flying up at you, the demand will fail unless your assembly is also granted the permission to call to unmanaged code. Clearly something is broken if this is the state of affairs. The administrator shouldn't have to grant your assembly such a far reaching and generic privilege as `UnmanagedCode` just so that you can open a file. It's not like you're making random calls into unmanaged code.

The key statement in the last paragraph is that you're NOT making random calls into unmanaged code. You're calling through a trusted component (`FileStream`) that is designed to do one specific thing in unmanaged space - open a file for you. What needs to happen here is a conversion of a very generic demand (for `UnmanagedCode`) to a more specific and reasonable demand (for `FileIOPermission`). This is why the `Assert` method exists.

Here's how it works. Before making the call to unmanaged code on your behalf, the `FileStream` object first demands that you have permission to open the file. This means the `FileStream` object creates a `FileIOPermission` object, specifying the file and access mode you've requested, and calls `Demand()`. If this fails, the `SecurityException` propagates up to you and you're done. If it succeeds however, the `FileStream` creates a `SecurityPermission` object (one that specifies the `UnmanagedCode` security permission), and calls `Assert`. It then makes the call to `CreateFile`, and when the interop layer demands `UnmanagedCode`, the demand succeeds, because `Assert` halts the stack walk.

You can think of `Assert` as the opposite of `Deny` (figure 3.18 shows an example). `Deny` has the power to halt the stack walk, but results in access being denied. `Assert` can also halt the stack walk, but results in access being

granted. Assert is all about stating your own authority to do something. But asserting permissions is clearly not something to take lightly. Any assembly that calls Assert must itself be granted the SecurityPermission called Assertion, which mscorlib is granted via policy, and naturally you cannot assert permissions that your own assembly doesn't have. If you've broken these rules in your call to Assert, the Assert call itself will generate a SecurityException.

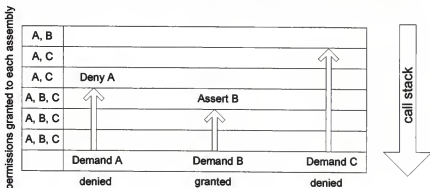


Figure 3.18: Deny versus Assert in a stackwalk

A stack frame starts with none of these markers, and you can restore yourself back to this state by calling one of several static methods on `CodeAccessSecurity`, as shown in figure 3.19.

```
// Excerpt from class CodeAccessPermission
class CodeAccessPermission {
    public static void RevertDeny();
    public static void RevertPermitOnly();
    public static void RevertAssert();
    public static void RevertAll();
}
```

Figure 3.19: How to remove CAS stack frame modifiers

Summary

- CAS provides a component-aware security model
- Security policy can be configured per machine, user, or host
- Loader discovers evidence when loading an assembly
- Evidence is input to policy, result is permissions for an assembly
- CASPOL used to read and write policy
- Demand causes a stack walk to prevent luring attacks
- Assert is used to convert generic demands to more specific ones

Module 4

Managed Type Fundamentals

The Common Language Runtime deals in types and instances of types. This is true no matter what programming language or technology is used to interface with the CLR. A type definition may have members that influence the way the type works as well as the way other developers interact with the type.

After completing this module, you should be able to:

- package code into CLR-compliant types
- encapsulate implementation details using private or internal members
- use and define properties and indexers
- use interface-based programming techniques
- understand the role of base classes in the CLR

Managed Type Fundamentals

The Common Language Runtime deals in types and instances of types. This is true no matter what programming language or technology is used to interface with the CLR. A type definition may have members that influence the way the type works as well as the way other developers interact with the type.

The role of type

Types are the fundamental unit of design, implementation, and execution in .NET

- Programmer describes types in code
- CLR-based compilers map language abstractions into CLR type definitions in terms of CIL
- CLR manages type instance layout/lifecycle at runtime
- Types package related code into reusable abstractions
- All objects and values are instances of types
- Several well-known base types and attributes distinguish structs, enums, types, interfaces

The Common Language Runtime exists to manage type definitions. The runtime can load types, execute methods of a type, and instantiate objects and values of a type. The existence of type is retained both in the underlying executable format as well as the execution of a running program. While the CLR does support fields and methods that are not enclosed in a type, at least two languages (Visual Basic and C#) do not support defining such members.

Virtually all code and data in the CLR is affiliated with a type. While programming languages may mask this by introducing their own concepts of types such as class, interfaces, records, modules, packages, structures, enumerations, and delegates, ultimately all of these forms are represented as type definitions at the lowest levels of the runtime. What distinguishes these types to the runtime is either the presence of well-known attributes in the metadata and/or the presence of well-known base types. For example, consider the component source code shown in figure 4.1. These five types were defined using the `class`, `interface`, `struct` and `enum`, and `delegate` constructs in C#, each of which has its own unique lexical syntax. However, ILDASM shows that each of these types has a rather similar appearance in the metadata, as shown in figure 4.2.

```
public class      TheClass {}
public interface TheInterface {}
public struct    TheStruct {}
public enum      TheEnum { }
public delegate void TheDelegate();
```

Figure 4.1: Type definitions in C#

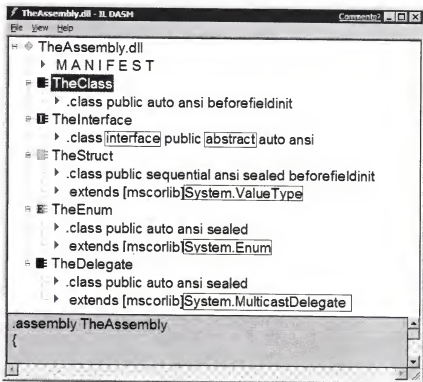
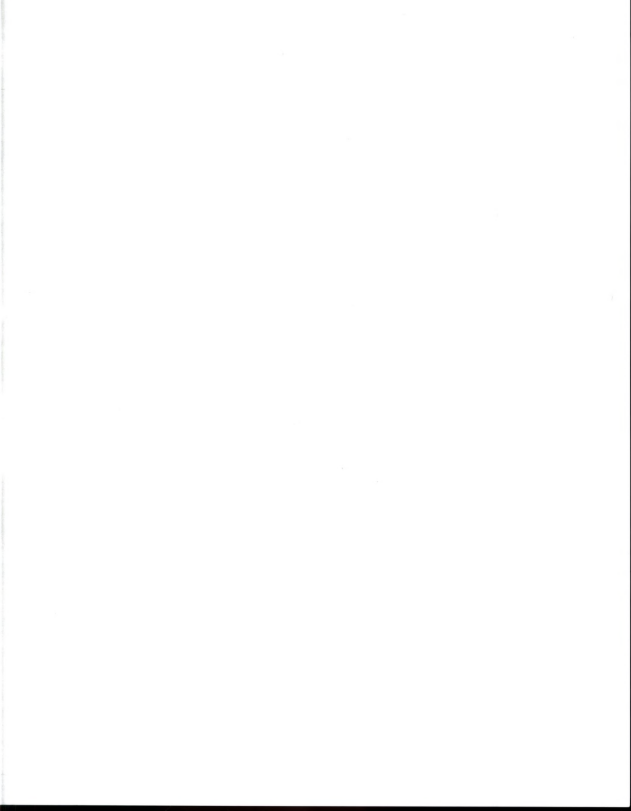


Figure 4.2: Distinguishing types via flags and base types



Access control

Access to types and their members is enforced by both compilers and the CLR

- Access to types can be restricted to intra-assembly
- Access to type members can be restricted to intra-type or intra-assembly
- Access modifiers specified on each member

A CLR type definition contains a set of member declarations. A member declaration describes one aspect of the state or behavior of the type. The CLR supports six different kinds of members. Nested types are members that provide implementation support for their enclosing type. Fields are associated with storage and are used to hold values or references of a given type. Methods are associated with code and represent operations that can be performed on a type. Constructors are special methods that are invoked by the CLR at initialization-time. Properties and events are metadata hints that describe the intended use of one or more methods.

Of the six kinds of members, fields, methods, and nested types are the three core member kinds. Constructors are structurally just methods with a well-known name. Properties and events, on the other hand, are additional annotations on a type that bind one or more "hidden" methods to a particular usage.

All members of a type are accessible to methods of that type. Access from methods of all other types is controlled using access modifiers. The list of access modifiers is shown in figure 4.3. In C#, if no access modifier is specified, the member is treated as if it were declared as `private`. Members that are `private` can only be accessed by methods of the type in which they are declared. This supports the classic C++ notion of encapsulation. In the CLR (as in COM before it), the preferred encapsulation boundary is a component, which in the case of the CLR is an assembly. To mark a member as accessible from anywhere in the current component, the `internal` access modifier must be used. To turn off access controls altogether, the `public` modifier should be used. Members marked as `public` accessed by methods of any type, independent of where the type is defined. The `protected` access modifier grants access to types that derive from the declaring type, again, to support C++-style encapsulation. The `protected` modifier may be combined with the `internal` modifier, granting access to any type that is either in the same assembly as the declaring type or that derives from the declaring type. Figure 4.4 shows some C# code that describes three types, one of which (`Foo`) has members.

	C#	VB.NET	Meaning
Type	public	Public	Type is visible everywhere
	<u>internal</u>	<u>Private</u>	Type is only visible inside of declaring assembly
Member	public	Public*	Member is visible everywhere
	internal	Friend	Member is only visible inside of declaring assembly
	protected	Protected	Member is only visible inside of declaring type and its subtypes
	protected internal	Protected Friend	Member is only visible inside of declaring type and its subtypes or other types inside of assembly
	<u>private</u>	<u>Private*</u>	Member is only visible inside of declaring type

* VB.NET defaults to **Public** for methods and **Private** for fields declared using the **Dim** keyword

Figure 4.3: Access modifiers

```
public class PubType
{
}

internal class IntraAssemblyType
{
}

class Foo // default access
{
    public void Foo() {}
    protected void SomeMethod() {}
    int w; // default access
    internal int x;
    protected internal int y;
    private int z;
}
```

Figure 4.4: Access modifiers

Independent of what access modifiers are applied to a given member, methods of the declaring type may access the member. Moreover, methods of nested types that are declared in the same type also may access **private** members of the surrounding type. However, the methods of the declaring type cannot access **private** members of the nested types they may declare.



Properties

A property is a named, typed member that provides field-like access syntax to the client, while providing method-like implementation control for the type developer

- Implementor defines "getter" and/or "setter" blocks of code to be called when client accesses property
- Compiler synthesizes method implementation(s) and relates them to named property using attributes
- Client compiler converts right-hand-side operations on property to getter method call
- Client compiler converts left-hand-side operations on property to setter method call
- Changing from field to/from property requires client recompile

The runtime supports three primitive kinds of members: nested types, methods and fields. For historical reasons, many programmers like the syntactic simplicity of using public fields. However, using public fields breaks the encapsulation of the declaring type and makes evolution difficult. The CLR provides properties as a way to provide the illusion of public fields while at the same time maintaining encapsulation.

In the CLR, a property is a binding of a name to one or two method declarations, one of which is the "getter," the other of which is the "setter." A property also has a type, which applies to the return type of the "getter" method and the last parameter to the "setter" method. Each language provides its own syntax for defining properties, but by the time the compiler has done its magic, properties have a standardized representation in the component metadata.

In C#, a property definition looks like a hybrid of a field declaration with scoped method definitions. Figure 4.5 shows the a type with three property definitions. The `Name` property only supports evaluation, therefore only has a getter method. Note that the implied return type of the getter is `string`, since that is the type of the property. In this case, the getter simply returns the value of a private field. Similarly, the `IsAdult` property only supports evaluation, although in this case, the return value is synthesized by analyzing some privately held state. A property, like a method, may use any means desired to determine the value to return to the caller. The `Phone` property in this example is both readable and writeable. Note that the argument being passed to the setter is accessed via the intrinsic `value` parameter, whose type is the same as the type of the property (`string` in this example). Figure 4.6 illustrates how a property is accessed by a client.

```
public class Person {
    private string name;
    private string phoneNum = "";
    private int age;

    // Constructor not shown...

    public string Name { // read-only property
        get {
            return(name);
        }
    }

    public bool IsAdult { // read-only property
        get { return(age >= 18); }
    }

    public string Phone { // read-write property
        get {
            return(phoneNum);
        }
        set {
            phoneNum = value;
        }
    }
}
```

Figure 4.5: Declaring a property

```
void Main() {
    Person p = new Person("Moe", 42);
    p.Phone = "555-1212";
    Console.WriteLine("You can reach {0} at {1}.", p.Name,
p.Phone);
    if( !p.IsAdult ) {
        Console.WriteLine("{0} is a minor.", p.Name);
    }
}
```

Figure 4.6: Using a property

Assuming a variable `obj` of type `TheType`, the expression `obj.TheProperty` is equivalent to the expression `obj.get_TheProperty()`. However, the `get_TheProperty` method definition is flagged with the `specialname` metadata attribute and programming languages such as C# or VB.NET will not allow you to access it directly, rather, the only way to invoke the method is through a property access. Similarly, attempting to treat `obj.TheProperty`

as the left-hand-side of an assignment triggers a call to the `set_TheProperty` passing in the right-hand-side of the assignment statement. Again, the `set_TheProperty` method cannot be directly invoked from C# or VB.NET.

Properties do not need to correspond to the fields of a type. Rather, they can be used to expose the public facade you wish the state of your instance to expose. Internally, property implementations may use calculation or lazy evaluation to return the appropriate value.

Properties are intended to simulate public fields. In that spirit, it is bad style to put important application behavior in a property definition. Rather, this is what methods are for. This is especially important for property getters, as it is not a tremendous leap to envision smart plumbing that caches the results of a property getter. Relying on a getter being called to enforce some application protocol is dangerous.

Visual Basic supported parameterized properties (also known as indexed properties). In essence, a parameterized property requires the caller to provide additional parameters in addition to the parameter name. The future of parameterized properties in general is unknown given that C# does not support them in the general case.

Indexers

Indexers are parameterized properties that allow clients to treat the type instance as an array.

- Declared like a property in C#, but always named `this`
- Getter/setter implementation syntax the same as properties
- Clients use array-like access syntax: `objref[param(s)]`
- May be overloaded based on parameter type or number (just like a method)

Indexers allow an instance to be accessed using array element access syntax. Each language provides its own syntax for defining indexers. Figure 4.7 shows an indexer that makes an instance feel like an associative array whose internal fields are accessed dynamically by name. Figure 4.8 shows the indexer in action. Note that in C# (but not VB.NET), you cannot access the indexer by its name, but rather, you must access it as if the instance is an array. As with properties, the language specific syntax of indexer implementation is converted into a pair of getter/setter methods and property named `Item` by default.

```
public class PhoneBook {
    Person[] entries = ...;

    public Person this[string name] { // Access by name
        get {
            foreach( Person p in entries ) {
                if( p.Name == name )
                    return(p);
            }
            return(null);
        }
        set {
            for( int n = 0; n < entries.Length; n++ ) {
                if( entries[n].Name == name ) {
                    entries[n] = value; // Update entry
                    return;
                }
            }
            // Add new entry to private array...
        }
    }
    public Person this[int n] { // Used by telemarketers
        get {
            if( n < entries.Length )
                return(entries[n]);
            return(null);
        }
    }
}
```

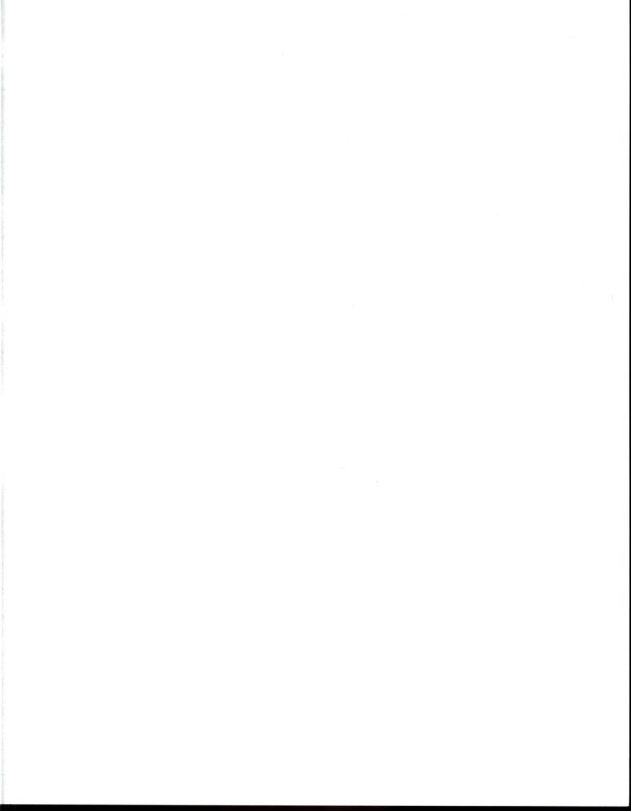
Figure 4.7: Declaring an indexer

```
class App {
    static void Main() {
        PhoneBook pb = ...;
        Person friend = pb["Mary"]; // getter
        Call(friend.Phone);

        pb["Moe"] = new Person("Moe", 42); // setter

        Person first = pb[0]; // getter
        Console.WriteLine( "First person in phonebook is {0} at
{1}",
                                first.Name, first.Phone );
    }
}
```

Figure 4.8: Using an indexer



Interfaces

Interfaces are an explicit kind of type in the CLR

- Express what is common across classes
- Allow classes to share a common design
- Include sufficient type information to program against, but not enough to instantiate
- Members may include methods, properties, indexers, and events
- Cannot include implementation details or fields
- Concrete type must supply all implementation details

Figure 4.9 shows a simple interface declaration (`ICowboy`) and two typical interface implementations (`Tex` and `Woody`). Notice that in the interface declaration, access modifiers are not used on the members - they are, by definition, public. Attempting to declare them otherwise (or even redundantly declare them public) results in a compiler error. Each implementation of the interface, on the other hand, declares each member to be a public part of the class signature.

```
interface ICowboy {
    void Draw();
    string Name { get; }
    object this[int n];
}

class Tex : ICowboy {
    public void Draw() { Console.WriteLine("Bang!"); }
    public string Name { get { return("Tex"); } }
    public object this[int n] { get { return(...); } }
}

class Woody : ICowboy {
    public void Draw() { Console.WriteLine("Bang!"); }
    public string Name { get { return("Woody"); } }
    public object this[int n] { get { return(...); } }
}
```

Figure 4.9: Declaring and implementing interfaces

Figure 4.10 demonstrates the use of this interface. Note that the `Tex` class is instantiated, and the resulting object referenced stored in a variable of type `ICowboy`. Also note that after the `Tex` object is used, the interface reference is reset to refer to an instance of the `Woody` class. This demonstrates the typical type compatibility rules for classes that implement interfaces.

```
ICowboy cb = new Tex();
cb.Draw();
Console.WriteLine("{0}, {1}", cb.Name, cb[0]);
cb = new Woody();
cb.Draw();
Console.WriteLine("{0}, {1}", cb.Name, cb[9]);
```

Figure 4.10: Using interfaces

Type Compatibility and Navigation

C# supports typical type compatibility handling while the CLR supports explicit runtime type navigation

- Types may implement multiple interfaces
- A class must statically declare which interface it supports
- The set of interfaces supported by a class applies to all instances of that class
- An object is type-compatible with interface *x* if and only if the object's class supports interface *x*
- C# *is*, *as*, and *typecast* operators support explicit run-time type navigation

Figure 4.11 illustrates how to navigate an object's type hierarchy at runtime using C#. Within the `DrawAndPaint` method, the object reference passed to this method is cast to the `IArtist` interface. If, at run-time, the CLR determines that the specified object does not in fact implement the `IArtist` interface, an exception `InvalidCastException` will be raised.

```
interface ICowboy {}
interface IArtist {}
class Tex : ICowboy {}
class LaRoche : IArtist {}

// What happens if DrawAndPaint is passed
// a Tex or LaRoche instance?
//
void DrawAndPaint( object o ) {
    // InvalidCastException on failure:
    IArtist a = (IArtist)o;
    a.Paint();

    // False on failure:
    bool IsArmed = o is ICowboy;

    // Null reference on failure:
    ICowboy cb = o as ICowboy;
    if( cb != null ) {
        cb.Draw();
    }
}
```

Figure 4.11: Type navigation operators

The second bit of syntax demonstrates the use of the `is` operator. Like the `typecast` operator, the `is` operator performs a run-time type check to see if the specified object implements the `ICowboy` interface. However, the result of this run-time check will simply be a boolean result indicating whether or not the specified object implements the requested interface.

Finally, the `as` operator is demonstrated. As with the `typecast` and `is` operator, `as` performs a run-time type check. However, the result of this operator is an object reference. If the requested type is in fact supported by the target object, a non-null object reference will be returned. If, however, the target object does not implement the specified interface, a null reference will be returned. The result of this expression is then tested using normal comparison expressions.

Explicit Interface Implementation

Explicit implementation syntax resolves member name clashes in multiple supported interfaces

- Hides members from class signature (members must be private)
- Interface entry points available only via interface reference

Figure 4.12 demonstrates the canonical use of C#'s explicit interface implementation syntax. In this example, the `LaTeX` class advertises support for both the `ICowboy` and `IArtist` interfaces. However, since both interfaces have a method named `Draw` with the identical signature, the `LaTeX` class cannot simply declare the `Draw` method public in its class declaration. Attempting to declare two different `Draw` methods would result in a compiler error, since a type cannot have two identical members, and (even if it did) no client compiler could possibly tell based on a call to `Draw` which method was the intended target.

```
interface ICowboy { void Draw(); }
interface IArtist { void Draw(); }

class LaTeX : ICowboy, IArtist
{
    void ICowboy.Draw() { Console.WriteLine("Bang!"); }
    void IArtist.Draw() { Console.WriteLine("Brush, brush"); }
}

LaTeX starvingDrifter = new LaTeX();
starvingDrifter.Draw(); // Compiler error - no public
Draw()
ICowboy cb = starvingDrifter;
cb.Draw();              // Bang!
IArtist a = starvingDrifter;
a.Draw();               // Brush, brush
```

Figure 4.12: Explicit interface implementation

Instead, the `LaTeX` class prefixes the name of each `Draw` member implementation with the name of the interface for which this particular method satisfies the implementation. Also note that each of the `Draw` methods is declared (implicitly) to be private. Together, these two syntactic mechanisms allow the `LaTeX` class to indicate exactly which `Draw` method a particular piece of code is associated with (the one on the `IArtist` interfaces, versus the one on the `ICowboy` interface). And as a result of making these members private, the `Draw` method will not be accessible via an object reference whose type is `LaTeX`. To access the `Draw` method, the client must explicitly select (using either the typecast or `as` operator) the interface whose `Draw` method they wish to invoke.

Base Classes

Every type has at most one base type

- `System.Object` and interfaces have no base type
- `System.Enum` for enums, `System.Array` for arrays
- Base type for classes defaults to `System.Object` if not explicitly specified
- sealed class modifier prevents use as a base type
- abstract class modifier mandates derivation

Structures are sealed by def.

All CLR types have at most one base type. In the CLR, base types provide a minimum set of fields and methods as well as triggering runtime semantics. For example, what makes a `struct` special in the runtime is that it is a type whose base type is the well-known type `System.ValueType`. The presence of this base type triggers a different code generator to handle variables of that type. Additionally, all `structs` inherit any fields or methods declared in `System.ValueType`.

The CLR provides several base types as part of the core type system. In particular, `System.Object`, `System.ValueType`, `System.Enum`, and `System.Array` are types that are not intended for direct instantiation but rather to act as bases for user-defined types to trigger different runtime behaviors. When programming language designers decide to invent type constructs such as `class`, `struct`, or `enum`, the compiler simply emits class definitions that specify the correct built-in base type (e.g., `System.Object`, `System.ValueType`, and `System.Enum`).

There are exactly two kinds of types that do not have base types in the CLR: interfaces and `System.Object`. `System.Object` is the root of the type system in the CLR and all objects and values are compatible with `System.Object`. If `System.Object` had a base type, then by definition there would be some objects or values that were not affiliated with `System.Object`, which would defeat the purpose of `System.Object` being the "universal" base type. The reason interfaces have no base type is that they only act as prototypical skeletons and never contain method implementation code. Rather, interfaces are only used to categorize classes, not to wholly or partially instantiate objects.

Programming languages typically allow programmers to manually set the base type of a class. In C#, the syntax for specifying a base type is shown in figure 4.13. When present, the base type must be the first in the list of compatible types. If the first type in list is an interface and not a class, then the implicit base type will be `System.Object`.

```
public class MyClassName : BaseImpl, Itf1, Itf2
{
    // member definitions go here
}
```

Name of base type

List of supported interfaces

Figure 4.13: Specifying a base class in C#

By default, a class may be used as a base type and may be used to instantiate objects. It is possible to suppress either usage using either class modifiers or by using access modifiers on the constructor. The former is the most direct way, the latter is more tricky than anything else.

A class that is declared as `sealed` cannot be used as a base type. Declaring a class as `sealed` ensures that no one can derive from your type and tamper with your method implementations. For example, if your implementation of `ICloneable.Clone` is particularly sensitive (they usually are), your class should be `sealed`. If your class is not `sealed`, anyone can declare your class as a base type and replace your `Clone` method with their own. Additionally, a non-`sealed` class cannot prevent derived types from adding additional interfaces to its list. This can be a problem when marker interfaces are used, as the base class may have explicitly omitted a critical marker interface, which the derived type is now free to add. In general, writing a class that can be used as a base requires considerable extra attention at both the design and coding phase. If you aren't willing to put in the effort to prevent these kinds of problems, you are better off declaring your classes as `sealed`.

A class can also be declared as `abstract`. An abstract class can only be used as a base type and cannot be used to instantiate objects. An abstract class usually contains one or more abstract method declarations, but it is not required to. However, if a class has one or more abstract method declarations, it must explicitly be declared as an abstract class. Defining an abstract class is generally easier than defining a non-`sealed`, non-abstract class, as in the latter case, you are focused as much on the base-derived contract as you are on the public contract.



Base/derived construction

Constructors and base types have "issues"

- Base type constructors not part of derived type's signature
- Base type constructors can be called from derived constructors using C++-like syntax
- Overloaded constructor on self can be called using C++-like syntax
- Base type's constructor executes using the most-derived type (like Java, not C++)
- Waterfall construction model for C# allows derived members to be accessed prior to initialization

Strictly speaking, the constructors of the base type are not inherited. You cannot use a base type constructor to instantiate a derived type. Rather, it is up to the derived type to provide one or more constructors to support instantiation. If the derived type does not declare a constructor, the compiler will typically generate a default (no-arg) constructor that calls the base type's default constructor. If the base type does not have a default constructor (or if it is not accessible to the derived type due to its access modifier), then the constructor-less derived class is invalid.

It is critical that the derived class's constructor call the base type's constructor. In C#, if you fail to do this explicitly, the compiler will automatically generate a call to your base type's default constructor (which must be accessible to the derived type). You can call the base type's constructor explicitly using the syntax shown in figure 4.14.

```
interface IPerson {...}
interface ICowboy {...}

public class PersonImpl : IPerson {
    public PersonImpl(string name) {...}
    public PersonImpl(string name, int age) {...}
}

public class Cowboy : PersonImpl, ICowboy {
    public Cowboy(string n, int a) : base(n, a) {...}
    public Cowboy(string n) : base(n) {...}
    public Cowboy() : this("Tex", 34) {}
    public Draw() {...}
}
```

Figure 4.14: base types and constructors

When a base type is present, the C# compiler emits consistent if not intuitive code for a constructor. The emitted code will first call all field initializers in order of declaration. This applies only to instance fields declared in the derived class. Once the derived type's field initializers have been called, the derived constructor calls the base type constructor, using the programmer-provided parameters if the base construct was used. Once the base type's constructor has completed execution, the derived constructor resume execution at the body of the constructor (i.e., the part of the constructor in braces). This means that when the base type's constructor executes, the derived type's constructor body has not even begun to execute. This is shown in figure 4.15.

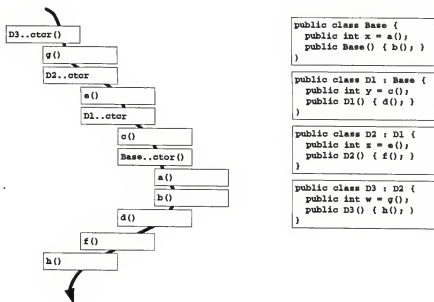


Figure 4.15: Derivation and construction

Because the base type's constructor is always called from the derived type's constructor, the actual type of the object seen by the base type constructor is not that of the base type but rather the derived type. This means that performing type-specific operations (like calling `GetType` or invoking virtual methods) will use the most-derived type of the object to locate the appropriate implementation. This means that it is possible for a derived type's virtual methods to execute before the derived type's constructor has completed. This is consistent with the characteristics of the Java virtual machine but is contrary to the behavior of C++. Figure 4.16 shows a program that ignores the pitfalls of base type construction. Note that in this program, the base type constructor causes two virtual methods to be called. In both cases, the derived implementation expects that the body of the constructor has executed. Because in this case it hasn't, the initial value of the field will be zero, which causes the derived type to malfunction.

```
using System;

public class Base {
    protected int GetLevel(int id) { return 42; }
    public virtual void DoItNow() { }
    public Base() {
        Console.WriteLine(this); // calls derived's ToString
        this.DoItNow();           // calls derived's DoItNow
    }
}

public class NuclearReactor : Base {
    private int x;
    public NuclearReactor(int id) { x = GetLevel(id); }
    public override void DoItNow() { Debug.Assert(x > 2); }
    public override string ToString() {
        if (x < 3)
            return("run for your lives");
        else
            return("don't worry, be happy");
    }
}
```

Figure 4.16: Perils of base construction

Method Invocation

Modifiers control dispatching and base/derived relationships

- Unadorned methods bound statically at compile-time based on the type of the object reference
- Modifiers enable dispatching based on the run-time type of object (not the reference)
- Base classes use modifiers to indicate whether derived types may, must, or may not replace the base class implementation
- Derived classes use modifiers to indicate whether a method replaces, or is unrelated to, a base method with the same name and signature
- Some modifiers can be combined to refine base/derived relationship

By default, a class method declared with no modifiers (and that does not satisfy the implementation of an interface method) will be bound statically at compile time. This means that when the compiler processes a client's call to such a method, it will use the type of the client's object reference to determine that target location for the call instruction. No run-time dispatching will occur.

Figure 4.17 illustrates the C# keywords that affect the binding from a client callsite to a target method. These modifiers are logically grouped into two sets, depending on the role the developer is taking on at the time. From the perspective of a base class designer, where the developer is trying to indicate what future derived classes may or may not be allowed to do, the *virtual*, *abstract*, and *sealed* modifiers are used. These three modifiers allow the base class designer to decide whether or not future derived classes may (*virtual*), must (*abstract*), or may not (*sealed*) override and replace the base class implementation with their own implementation. Calls to *virtual* and *abstract* methods will be bound dynamically at run-time based on the type of the object the client's object reference actually refers to.

From the perspective of a base class designer	
C# Syntax	Meaning
virtual	Method may be replaced by derived class.
abstract	Method must be replaced by a derived class (or redeclared abstract).
sealed	Method may not be replaced by a derived class.

From the perspective of a derived class designer	
C# Syntax	Meaning
override	Method replaces a virtual/abstract/override method in the base with its own implementation, and is still overridable by further derived types.
new	Method is unrelated to a similarly defined method in the base class. Typically used to deal with a change to a base class that creates a collision where one didn't exist before, and where it's not practical to just choose a different method name.

Figure 4.17: C# method modifiers

From the perspective of a derived class designer, the *override* and *new* modifiers are used to inform the compiler about the relationship of the method being declared to an identical method declaration that appears in the base class. The *override* modifier is used to indicate the developer's intent to replace a base class *virtual* or *abstract* method with their own more-derived

implementation. The `new` modifier, which will be used infrequently in practice, is used to indicate the developer's desire to tell the compiler that the method she's declaring is unrelated to the base class method of the same name and signature (and that, if this method is also virtual, a "new slot" in the virtual function dispatch table should be setup - hence the name of the modifier). In practice, this modifier will be used when you have a lot of code written against a particular function in the derived class and, later in the development cycle, the base class introduces a method with the exact same name and signature. In this situation, when the derived class is compiled, the compiler will start issuing a warning about the potentially confusing situation. When this happens, the derived class developer has two options: (1) rename the derived-class method so that there is no longer a conflict or (2) use the `new` modifier. The first solution is often problematic, because this also means all client calls to the original derived class method will need to be recoded to refer to the newly named method. For this reason, the `new` modifier is useful, since maintains the binding from the original calls to the derived class method without requiring clients to be recoded.

Figure 4.18 demonstrates the use of these modifiers to specify the relationship between three classes, with source code comments indicating the result.

```
abstract class Base {
    public void a() {} // statically bound, cannot
    override
    public virtual b() {} // may be overridden
    public abstract c(); // must override or redeclare
    abstract
    public virtual d() {} // may be overridden
}

class Derived : Base {
    public override a() {} // illegal: Base.a not overridable
    public override b() {} // legal: Base.b override allowed
    public override c() {} // legal: Base.c override required
    public new d() {} // unrelated to Base.d, not
    overridable
}

class MoreDerived : Derived {
    public override c() {} // legal: Derived.c still
    overridable
}
```

Figure 4.18: Using method modifiers

Figure 4.19 shows which method modifiers may be combined, and the resulting semantics.

after using new, how can you call a base class?

Method modifier combinations	
C# Syntax	Meaning
sealed override	This method replaces a virtual/abstract/override method in the base with its own implementation, and terminates the overridability of this method as far as further derived types are concerned.
new virtual	This method is unrelated to a similarly defined method in the base class and gets a new slot in the associated virtual method dispatching table for this class. Further derived types may choose to override this method.
new abstract	This method is unrelated to a similarly defined method in the base class and gets a new slot in the associated virtual method dispatching table for this class. Further derived types must to override this method.

Figure 4.19: Combining method modifiers

Figure 4.20 shows a revised piece of sample code using combinations of method modifiers, with source code comments indicating the result.

```

abstract class Base {
    public void a() {}    // statically bound, cannot
    override
    public virtual b() {} // may be overridden
    public abstract c() {} // must override or redeclare
    abstract
    public virtual d() {} // may be overridden
}

abstract class Derived : Base {
    public sealed override b() {} // terminal override of
    Base.b
    public new virtual d() {}    // unrelated to Base.d,
    overridable
}

class MoreDerived : Derived {
    public override b() {}    // illegal: Derived.b
    sealed
    public override c() {}    // overrides Base.c
    public override d() {}    // overrides Derived.d
}

```

Figure 4.20: Combining method modifiers

Summary

- Types are the fundamental currency in the CLR
- Access to types may be controlled
- Access to a type's members may be controlled
- Properties and indexers bind get/set methods to a single name
- Managed types may implement multiple interfaces
- Managed types may only have one base type

Module 5

Reflection and Attributes

The Common Language Runtime makes virtually every facet of a type definition available to programmers in any language. The presence of such rich type information, as well as an API for accessing it easily at runtime, enable numerous forms of tool development, and type system interoperability.

After completing this module, you should be able to:

- ❑ load types and invoke methods without a priori knowledge
- ❑ create user-defined attributes to express application/organization-specific constraints
- ❑ discover attributes at runtime

Reflection

The Common Language Runtime deals in types and instances of types. This is true no matter what programming language or technology is used to interface with the CLR. A type definition may have members that influence the way the type works as well as the way other developers interact with the type.

Reflection

The Common Language Runtime makes type information ubiquitous, accessible, and extensible

- Reflection allows anyone to see any type (barring security)
- Reflection information extensible via custom attributes
- Reflection information never optional - all things are knowable
- `System.Reflection` namespace root of library support
- Enables numerous runtime-provided services (remoting, serialization, etc.)
- Enables numerous kinds of tool development (documentation, code generation, etc.)

Types are described using a standard metadata format that is documented as part of the CLI. To enable a broader range of programmers to take advantage of metadata, the CLR provides a metadata parser and generator to make metadata available at runtime as well as at development time.

The metadata for a type retains as much of the programmer's intention as is possible. In general, however, the programmer's intention is not to specify the in-memory representation of instances or supporting data structures. To this end, the CLR uses a virtualized layout scheme in which in-memory representations are determined as code is loaded and initialized. This is a radical departure from classic C/C++ development, where in-memory representations are hard-coded into the instruction stream, causing brittleness when working with independent compilation units.

The fact that all aspects of a type definition are available via metadata enables services (and programming models) to emerge that are not possible otherwise. Accessing a type's metadata at runtime is called reflection and is exposed via the `System.Reflection` library. Reflection is a key enabling technology for programs that write programs. Most modern IDE's depend on reflection. Most XML and DBMS integration depends on reflection. Most RPC/remote method invocation technology relies on reflection. Reflection is also useful for writing programs to automate tedious coding practices. This is illustrated in figure 5.1.

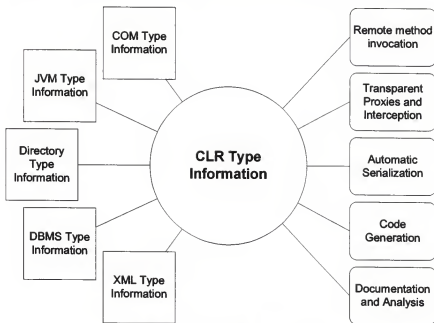


Figure 5.1: The role of reflection

Figure 5.2 demonstrates the canonical use of reflection. This example generates a SQL CREATE TABLE statement based on the schema of an object. Note that this generative program can be run against any CLR-based type. Also note that if defects are found in the generated code, this routine can be repaired and rerun, allowing the defect to be fixed once, not multiple times.

```
public static void GenerateSQL(Object obj) {  
    Type type = obj.GetType();  
    Console.WriteLine("create table {0} (", type.Name);  
    bool needsComma = false;  
    foreach (FieldInfo field in type.GetFields()) {  
        if (needsComma) Console.WriteLine(",");  
        else needsComma = true;  
        Console.WriteLine("{0} {1}", field.Name, field.FieldType);  
    }  
    Console.WriteLine(")");  
}
```

Figure 5.2: Using reflection

The reflection facilities of the CLR are unique in that they are completely extensible. It is possible to extend how CLR types work by defining new

semantics that cross-cut type hierarchies and express them via custom attributes. These attributes are themselves instances of CLR types that augment metadata constructs such as field, method, parameter and type declarations.

System.Type

`System.Type` is the focal point of reflection

- All objects and values are instances of types
- Can discover type of object or value using `GetType` method
- Can reference type by symbolic name using C# `typeof` keyword
- Types are themselves instances of type `System.Type`
- Inheritance/compatibility relationships traversable through `System.Type`

All type definitions are available at runtime and are used extensively in CLR-based libraries. Type definitions can be acquired in multiple ways. The simplest way to acquire a type definition at runtime is to use a compiler facility that maps the symbolic name of a type to its runtime representation. In C#, this is exposed using the `typeof` keyword. The `typeof` keyword accepts a symbolic type name and returns a reference to the corresponding type definition. It is also possible to ask any object or value for its type definition by calling the `GetType` method. The `GetType` method accepts no parameters and also returns a reference to the type description used to instantiate the object or value. As this method is exposed from `System.Object`, it is available for all objects and values. Figure 5.3 shows an example of both `GetType` and `typeof` being used on an instance of a class as well as all of its fields.

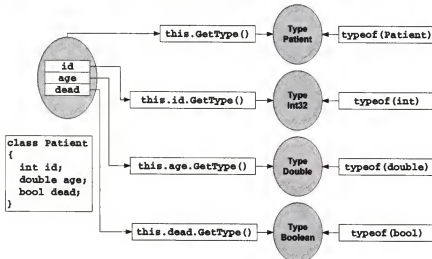


Figure 5.3: Pervasive type and `GetType`

The reference returned by `typeof` and `GetType` is of type `System.Type`. `System.Type` makes the underlying metadata for the type accessible to programmers working in any programming environment without concern for the underlying on-disk representation of the metadata or code. `System.Type` provides extensive information about the type, including its name, its containing module and assembly, a list of compatible types, and a list of its members. The relationship between the type objects themselves and the object that represents `System.Type` is shown in figure 5.4.

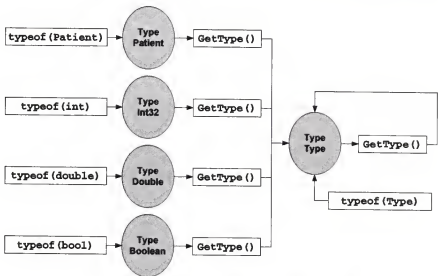
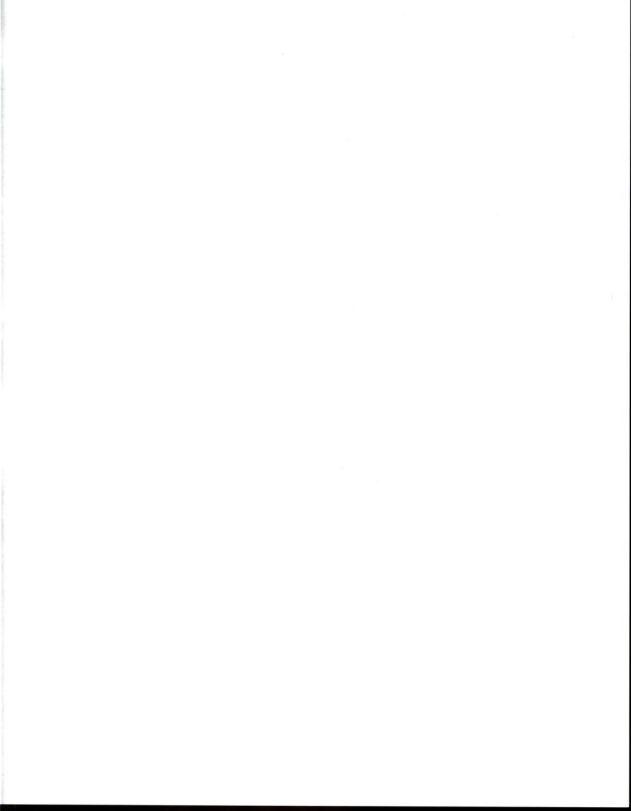


Figure 5.4: Pervasive type and System.Type



Using System.Type

All facets of a type are available at runtime

- Each kind of member has a reflection class that represents it
- Most types derive from `System.Runtime.MemberInfo`
- Can look up members by kind or globally
- Support for overloading and case-insensitive names for script/VB
- Can see non-accessible members if security allows

Given a `System.Type` object, you can discover all of the members of the type. Each element in the type graph is an instance of a given reflection type from the `System.Reflection` namespace. Figure 5.5 shows the type hierarchy for `System.Reflection`. The object model that is used in reflection is shown in figure 5.6. The `Assembly` is the root of the object graph. An `Assembly` consists of one or more `Modules`. A `Module` consists of one or more `Types`. A `Type` consists of one or more `MemberInfos`, where a member is either a field, method, constructor, nested type, property, or event.

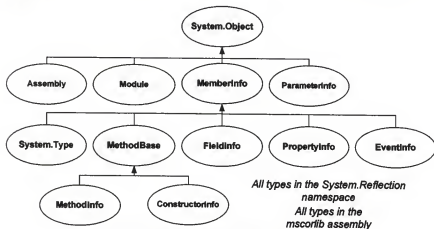


Figure 5.5: Reflection and the CLR type system

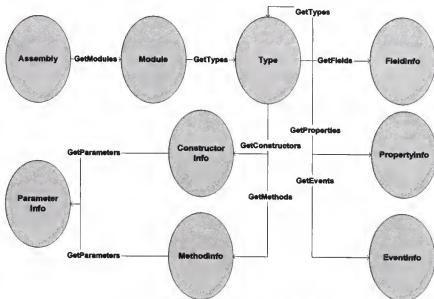


Figure 5.6: Reflection object model

Figure 5.7 shows the code to enumerate all members in an assembly. Note that the generic `GetMembers` method was called on the type to discover each of its members. As shown in figure 5.5, there is an abstract base type `MemberInfo` from which all 6-member types derive. The `MemberInfo` base class provides uniform access to the name of the member, the type in which it was declared as well as the type from which it was reflected. These latter two types will differ when a member becomes visible implicitly from a base type.

```
using System.Reflection;

public static void ListAllMembers(String assemblyName) {
    Assembly assembly = Assembly.Load(assemblyName);
    foreach (Module module in assembly.GetModules())
        foreach (Type type in module.GetTypes())
            foreach (MemberInfo member in type.GetMembers())
                Console.WriteLine("{0}.{1}", type, member.Name);
}
```

Figure 5.7: Walking every element in an assembly



Late Binding

Types may be instantiated and/or members accessed in a late bound manner

- Can instantiate type in memory, choosing constructor to call
- Can read/write fields of an object
- Can invoke methods
- Can invoke property getters and setters
- Public members always accessible
- Non-public members accessible if callers hold sufficient CAS permissions

Activator.CreateInstance

`Activator.CreateInstance` is the late-bound equivalent to operator `new`

- Allocates storage for new type instance
- Calls specified constructor
- Returns generic object reference
- Combined with interface-based member access provides flexibility and performance

Figure 5.8 demonstrates using reflection to perform a late-bound instantiation of a type, while still accessing the resulting object via a strongly typed interface reference. Figure 5.9 demonstrates how to pass arguments to a constructor using `Activator.CreateInstance`.

```
interface ICowboy { void Draw(); }
class Tex : ICowboy { ... }
class Woody : ICowboy { ... }

class App {
    static void Main() {
        Console.WriteLine("Enter western type to use: ");
        string typeName = Console.ReadLine();

        // Late-bound activation
        ICowboy cb = (ICowboy)
            Activator.CreateInstance(Type.GetType(typeName));

        // Early-bound member access:
        cb.Draw();
    }
}
```

Figure 5.8: Using reflection to instantiate a type

```
interface ICowboy { void Draw(); }
class Tex : ICowboy
{
    public Tex( string name, int NumGuns )
    {
        ...
    }
}

class App {
    static void Main() {
        object args[] = { "Jesse James", 4 };
        ICowboy cb = (ICowboy)
            Activator.CreateInstance(typeof(Tex), args);

        cb.Draw();
    }
}
```

Figure 5.9: Passing constructor arguments to `CreateInstance`

FieldInfo

FieldInfo can be used to set/get fields

- Field description returned by `Type.GetField`
- Can see name, type, accessibility and all other aspects of field declaration
- Can call `GetValue/SetValue` to access the field of an object or type
- Cannot see field initializers from C# (they are part of IL in constructor)

The simplest of the member types to deal with is the `FieldInfo` member, which provides information (and access) to a field declared in a class. As figure 5.10 shows, one can call `GetField` on a `System.Type` object to look up a field by name. Once the `FieldInfo` is found, it can be used to set or get the field value for a given object. Figure 5.11 shows an example of this.

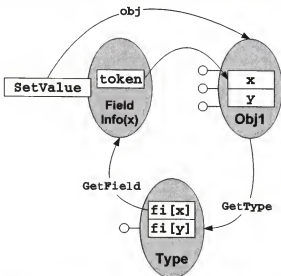


Figure 5.10: `System.Reflection.FieldInfo`

```
public static void SetX(Object target, int value) {
    // Write to field: int x
    Type type = target.GetType();
    FieldInfo field = type.GetField("x");
    field.SetValue(target, value);
}
```

Figure 5.11: Using `System.Reflection.FieldInfo`

MethodInfo/ConstructorInfo

MethodInfo/ConstructorInfo can be used to invoke methods/constructors

- Both types derive from common `MethodBase`
- Method description returned by `Type.GetMethod`
- Can see name, accessibility, return type, parameters and all other aspects of methods
- Can call `Invoke` to access the method of an object or type
- Also accessible from `System.Exception` as well as remoting plumbing

It is also possible to find descriptions of methods by calling the `GetMethod` method on `System.Type`. As shown in figure 5.12, there is a `MethodInfo` object for each method on a class. A `MethodInfo` can be used to find out about the name, return type, and argument types of the method. A `MethodInfo` object can also be used to indirectly invoke a method on an object. Figure 5.13 shows this usage by invoking a method named "Add" on an arbitrary object.

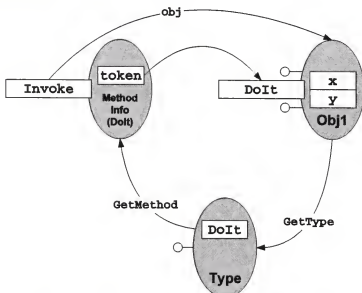
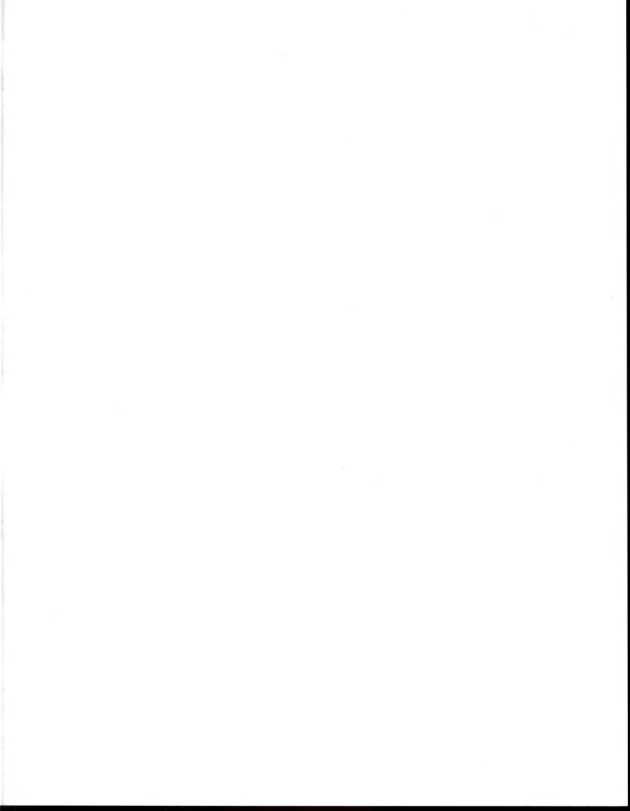


Figure 5.12: Using `System.Reflection.MethodInfo`

```
double CallAdd(Object target, double x, double y)
{
    // Call method: double Add( double, double )
    Type type = target.GetType();
    MethodInfo method = type.GetMethod("Add");

    if( method != null ) {
        object[] args = { x, y };
        object result = method.Invoke(target, args);
        return (double)result;
    }
    return(0);
}
```

Figure 5.13: Using System.Reflection.MethodInfo



PropertyInfo

PropertyInfo can be used to invoke property getters and setters

- Property description returned by `Type.GetProperty`
- Can see name, type, accessibility and all other aspects of property declaration
- Can call `GetValue/SetValue` to access the field of an object or type

Invoking property getters and setters is similar in ways to both invoking a method and getting or setting a field value. As with fields, you use `Get/SetValue` method calls against the `PropertyInfo` object to access the target object's property. And similar to using `MethodInfo.Invoke`, you need to pass a parameter to the target setter. Figure 5.14 demonstrates calling a property setter.

```
void SetName(Object target, string newName)
{
    // Set property: string Name { set {...} }
    Type type = target.GetType();
    PropertyInfo prop = type.GetProperty("Name");

    if( prop != null ) {
        prop.SetValue(target, newName, null);
    }
}
```

Figure 5.14: Using `System.Reflection.PropertyInfo`

Extending type information

CLR type information is extensible using custom attributes

- Allows user-defined aspects of a type to be visible via reflection
- Custom attributes are classes derived from `System.Attribute`
- C# uses IDL-like syntax with [] prior to the definition of the target
- Can be comma-delimited or in independent [] blocks
- Attribute parameters passed by position or name
- Attributes can be applied to an assembly or module using special syntax
- Attributes discovered using `IsDefined` and `GetCustomAttributes`

*can use reflection to access
private members*

A key feature of the CLR is the fact that the type descriptions are extensible to adapt to applications that were never anticipated by the CLR architects. The way type descriptions are extended is via custom attributes. A custom attribute may be encoded into your source code during development, however, the more important aspect is that your compiler will emit them into the metadata in a well-known format so others can easily detect their presence.

Custom attributes are used extensively by the serialization service, the Win32/COM interop service, the XML serializer and customized services such as threading and synchronization. Arguably, the CLR was developed because COM lacked extensible type information.

Every language has its own way of allowing attributes to be applied. Figure 5.15 shows the C# syntax. Note the use of the [] to signal the presence of attributes. The attributes always precede their target, and may have both positional and named parameters. The positional parameters must precede the named parameters.

```
[assembly: Author("Larry")] // Applies to assembly

[ Author("Moe", "Hi") ]      // Applies to class
class MyClass
{
    // Attribute applies to method:
    [ Author("Curly", Contact="curly@stooges.com") ]
    void f() {
        Object obj = null;
        obj.ToString();
    }
}
```

Figure 5.15: Specifying an attribute

Custom attributes are ultimately just classes that extend `System.Attribute`. Figure 5.16 shows the definition of a new attribute named `Author`. Note that this attribute class has two constructors. To invoke the default constructor, one can simply apply the attribute like this: `[Author("Fred")]`, or `[Author("Fred", "My boss made me do it")]`. In addition to using the two constructors, one could also configure any public properties or fields like this: `[Author("Fred", Contact="fred@acme.com")]`. The named parameters must correspond to a public field or property on the attribute class, otherwise a compiler error will occur. Figure 5.15 shows a sample application of the attribute defined in figure 5.16.

```
using System;

public class AuthorAttribute : Attribute {
    public string Name;
    public string Comment;
    public string Contact = "";

    public AuthorAttribute( string n, string c ) {
        Name = n;
        Comment = c;
    }

    public AuthorAttribute( string n )
        : this(n, "")
    {
    }
}
```

Figure 5.16: Defining a custom attribute

The `GetCustomAttributes` method returns an array of attribute objects to allow interrogation of attribute fields and properties. Calling `GetCustomAttributes` causes the serialized attribute constructors to execute as the attribute objects are instantiated from the metadata prior to your code seeing them. The `GetCustomAttributes` method is overloaded to either take no parameters, in which case all attributes are returned, or to take a `System.Type` object, which indicates what type of attribute is desired. In both cases, an (potentially empty) array of `System.Object` references is returned. Figure 5.17 shows this method in action.

```
using System;
using System.Reflection;

void DocType( string asmName, string typeName )
{
    Type attrType = typeof(AuthorAttribute);

    Assembly a = Assembly.Load(asmName);
    if( a.IsDefined(attrType) ) {
        Console.WriteLine("Assembly has an author");
    }

    Type t = Type.GetType(string.Format("{0}, {1}", typeName,
asmName));
    object[] attrs = t.GetCustomAttributes(attrType, false);

    foreach( AuthorAttribute author in attrs ) {
        Console.WriteLine("Name:           " + author.Name);
        Console.WriteLine("Notes:           " + author.Comment);
        Console.WriteLine("Contact info: " + author.Contact);
    }
}
```

Figure 5.17: Retrieving attributes

Summary

- Type information is easily accessible and ubiquitous
- Type information allows you to do things at runtime
- Type information allows you to do things at development-time
- Type information is extensible via custom attributes

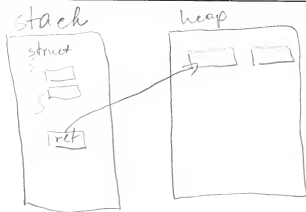
Module 6

Objects, Values, and Memory

Objects are instances of types that are managed by the runtime. Objects are allocated on a garbage-collected heap. Values are formatted bits of data that are allocated in-line with where they are declared (either on the callstack or embedded within an object in the heap).

After completing this module, you should be able to:

- ❑ understand the difference between value and reference types
- ❑ use and implement cloning
- ❑ understand the difference between pass-by-reference and pass-by-value
- ❑ correctly choose between reference and value types
- ❑ avoid excessive boxing overhead for value types
- ❑ define efficient data types
- ❑ understand the impact of object lifecycle, finalization and garbage collection



Objects, Values, and Memory

Objects are instances of types that are managed by the runtime. Objects are allocated on a garbage-collected heap and can be compared for both identity and equivalence. Values are simply formatted memory that does not incur the overhead of a distinct object.

Reference vs. value types

The runtime distinguishes between reference and value types

- Reference types yield distinct objects on the heap
- Value types yield instances whose lifetime is scoped by the declaring context
- In C# and VB, all classes are reference types
- In C# and VB, primitives, structs and enums are value types
- Value types cannot be bases to other types
- Value types cannot have "default" constructors
- Instances of value types do not have an object header (vptr)
- Instances of value types are not independently garbage collected

object header

The type system of the CLR distinguishes between types that correspond to simple values and types that correspond to more traditional "objects." The former are called "value types;" the latter are called "reference types." Value types support a somewhat limited subset of what a reference type supports. In particular, instances of value types do not carry the full overhead of a full-blown object. This makes value types useful in scenarios where the costs of an object would otherwise be prohibitive.

The term object is overloaded in the literature as well as in the CLR documentation. For consistency, we will define an object as an instance of a CLR type on the garbage collected (GC) heap. Objects support all of the methods and interfaces declared by their type. Objects always begin with the two-field object header described in the previous chapter to implement polymorphism. Value types (such as `int` or `bool`) are also CLR types, but instances of a value type are not objects, as they do not begin with an object header, nor are they allocated as distinct entities on the GC heap. This makes instances of value types somewhat less expensive than instances of reference types.

Reference types and value types are distinguished by base type. All value types have `System.ValueType` as a base type. `System.ValueType` acts as a signal to the CLR that instances of the type must be dealt with differently. Figure 6.1 shows one view of the CLR type system. Note that the primitive types such as `System.Int32` are descendants of `System.ValueType`, as are all user-defined structures and enumerations. All other types are reference types.

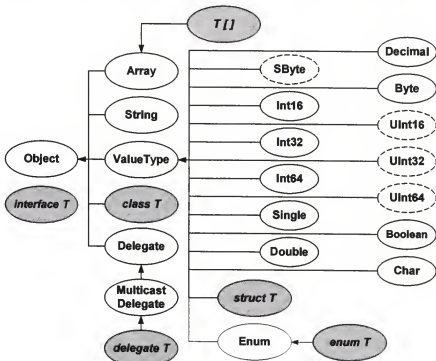


Figure 6.1: The CLR type system

Programming languages typically have a set of built-in or primitive types. It is the job of the compiler to map these built-in types to CLR types. The CLR provides a fairly rich set of standard numeric types as well as boolean and string types. Figure 6.2 shows the VB.NET and C# built-in types and the CLR types they correspond to. Note that all numeric types and boolean are value types. Also note that `System.String` is a reference type. That stated, in the CLR, `System.String` objects are immutable and cannot be changed once they are created. This makes `System.String` act much more like a value type than a reference type, as will be further explored later in this chapter.

CLR	C#	VB.NET	Size (bits)	Ref/Val
Object	object	Object	N/A	Reference
String	string	String	N/A	
Boolean	bool	Boolean	8	Value
Char	char	Char	16	
Single	float	Single	32	
Double	double	Double	64	
Decimal	decimal	Decimal	128	
SByte	sbyte	N/A	8	
Byte	byte	Byte		
Int16	short	Short	16	
UInt16	ushort	N/A		
Int32	int	Integer	32	
UInt32	uint	N/A		
Int64	long	Long	64	
UInt64	ulong	N/A		

Figure 6.2: C# and VB.NET built-in types

For a variety of reasons, value types cannot be used as base types. To that end, all value types are marked as sealed in the type's metadata and cannot declare virtual or abstract methods. Additionally, because instances of value types are not allocated as distinct entities on the heap, value types cannot have finalizers. These are restrictions imposed by the CLR. The C# programming language imposes one additional restriction, which is that value types cannot have default constructors. In the absence of a default constructor, the CLR simply sets all of the fields of the value type to their default values when an attempt is made to construct an instance of a value type. Finally, because instances of value types do not have an object header, all method invocation against a value type is non-virtual. This helps performance but loses some flexibility.

There are two ways to define new value types. One way is to define a type whose base type is `System.ValueType`. The other way is to define a type whose base type is `System.Enum`. In C#, the former is achieved using the `struct` keyword; the latter is achieved using the `enum` keyword.

A C# struct definition is strikingly similar to a C# class definition except for the choice of keyword. There are a few subtle differences, however. For one, you cannot specify an explicit base type for a C# struct; rather, `System.ValueType` is always assumed. You also cannot explicitly declare a C# struct as abstract or sealed; rather, sealed is implicitly added by the

compiler. Figure 6.3 shows a simple C# struct definition. Note that like a C# class definition, a C# struct can have methods, fields and support arbitrary interfaces.

```
public struct Size {  
    public int height;  
    public int weight;  
}
```

Figure 6.3: Defining a value type as a structure

C# structs are useful for defining types that act like user-defined primitives, but that contain arbitrary composite fields. It is also possible to define specializations of the integral types that do not add any new fields but rather simply restrict the value space of the specified integral type. These restricted integral types are called enumerations or enums.

Enumeration types are CLR value types whose base type is `System.Enum`. Enumeration types must specify a second type that will be used for the data representation of the enumeration. This second type must be one of the CLR's built-in integral types (excluding `System.Char`). An enumeration type can contain members, however, the only members that are supported are literal fields. The literal fields of an enumeration type must match the enumeration's underlying representation type, and act as the set of legal values that can be used for instances of the enumeration.

New enumeration types can be created using C# enum definitions. A C# enum looks similar to a C or C++ enum. As shown in figure 6.4, a C# enum definition contains a comma-delimited list of unique names. Each of these names will be assigned a numeric value by the compiler. If no explicit values are provided (as is the case in this example), then the compiler will assign the values 0, 1, 2, etc. in order of declaration. Unlike C, C# does not consider enumerations to be type-compatible with numeric types. Rather, to treat an enum like an int (or vice-versa), one must first explicitly cast to the desired type.

```
public enum Breath {  
    None, Sweet, Garlicy, Oniony,  
    Rancid, Horrid, Sewerlike  
}
```

Figure 6.4: Defining a value type as an enumeration

If no explicit underlying type is specified, the C# compiler will assume `System.Int32`. This default can be overridden using the syntax shown in figure 6.5. Also note that while the member names of an enumeration must be unique, there is no such uniqueness requirement for the integral values of each member. In fact, it is common to use enumeration types to represent bitmasks. To make this usage explicit, an enumeration can have the `[System.Flags]`

attribute. This attribute signals the intended usage to developers. This attribute also affects the underlying `ToString` implementation so that the stringified version of the value will be a comma-delimited list of member names rather than just a number.

```
[ System.Flags ]  
public enum Organs : byte {  
    None = 0,  
    Heart = 0x0001,  
    Lung = 0x0002,  
    Liver = 0x0004,  
    Kidney = 0x0008  
}
```

Figure 6.5: Defining a value type as a bitfield

Variables, parameters and fields

Memory is managed differently for value and reference types

- Reference type variables/params/fields are references to instances
- Value type variables/params/fields are instances
- Operator `new` allocates memory on heap for reference types
- Operator `new` simply invokes constructor for value types
- Assignment/initialization of values types yields second copy, not second reference
- Method parameters may be passed by value (default) or by reference independent of type
- Method declaration decides the policy per parameter using `ref/out` keywords in C#

Reference types always yield instances that are allocated on the heap. In contrast, value types yield instances that are allocated relevant to context in which the variable is declared. If a local variable is of a value type, the memory for the instance is allocated on the stack. If a field in a class is a member of a value type, then the memory for the instance is allocated as part of the layout of the object or type in which the field is declared. The rules for dealing with value and reference types are consistent for variables, fields and parameters. To that end, this chapter will use the term variable to refer to all three concepts and will use the term local variable when discussing variables by themselves.

As their name implies, reference type variables contain object references, not instances of the type they are declared as. A reference type variable simply contains the address of the object it refers to. This means that two reference type variables may refer to the same object. It also means that it is possible for an object reference to not refer to an object at all. Before a reference type variable can be used, it must first be initialized to point to a valid object. Attempts to access a member through an object reference that does not refer to a valid object will result in a runtime error. The default value for a reference type field is null, which is a well-known address that refers to no object. Any attempts to use a null reference will result in a `System.NullReferenceException`.

If the declared type of the reference is sealed, then the reference is guaranteed to refer to an instance of the declared type. If the declared type of the reference is not sealed, then the reference may refer to instances of types that are substitutable with the declared type. The rules for substitutability are different for class-based and interface-based references. Class-based references may refer to instances of the declared type or any type that derives directly or indirectly with the declared type. Interface-based references may refer to instances of any type that has declared compatibility with the declared type. It is possible that a class-based reference may refer to instances of the declared type. This is not possible for interface-based references, as all interface types are abstract.

Reference type variables require an object to do any meaningful work. In contrast, value type variables are the instances themselves, not references. This means that a value type variable is useful immediately upon declaration. Figure 6.6 shows an example of two types that are identical except that one is a reference type and the other is a value type. Note that the variable `v` can be used immediately, as the instance has already been allocated as part of the variable declaration. In contrast, the variable `x` cannot be used until it refers to a valid object on the heap. Figure 6.7 shows how the two variables are allocated in memory.

```

public struct Size {
    public int height;
    public int weight;
}
public sealed class CSize {
    public int height;
    public int weight;
}
static App {
    static void Main() {
        Size v;           // v is an instance of Size
        v.height = 100;    // legal
        CSize r;           // r is a reference
        r.height = 100;    // illegal, r is dangling
        r = new CSize();   // r refers to an instance of CSize
        r.height = 100;    // legal, r no longer dangling
    }
}

```

Figure 6.6: Using value and reference types

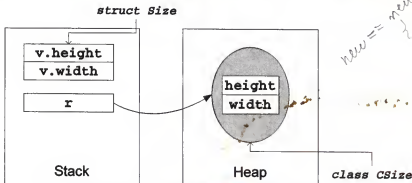


Figure 6.7: Reference and value types

It is interesting to note that the C# language allows you to use the `new` operator for both reference and value types. When used with a reference type, the C# `new` operator is translated to a CIL `newobj` instruction, which triggers an allocation on the heap followed by a call to the type's constructor. When a value type is used, the C# `new` operator is translated to a CIL `initobj` instruction, which simply initializes the instance in place using the default values for each field. In this respect, using `new` with a value type is similar to using C++'s placement operator `new` to invoke a constructor without allocating memory.

Assignment works differently for value and reference types. For reference types assignment simply duplicates the reference to the original instance, resulting in two variables that refer to the same object identity. For value types, assignment copies an instance from one variable to another, with the second copy completely unrelated once the assignment is done. Compare the code in figures 6.8 (illustrated by figure 6.9) and 6.10 (illustrated by figure 6.11). Note that in the reference type case, the assignment is only duplicating the reference, and that changes through one variable are visible through the other. In contrast, the assignment of the value type yields a second independent instance.

```
static App {
    static void Main() {
        CSize r1 = new CSize(); // r1 refers to instance of
        CSize
        CSize r2 = r1;          // r2 points to same object as
        r1
        r1.height = 100;
        r2.height = 200;
        bool truth = r2.height == r1.height;
        bool moreTruth = r1.height == 200;
    }
}
```

Figure 6.8: Using reference types

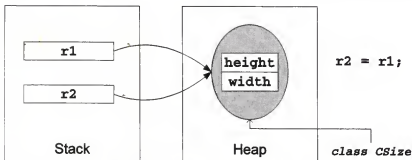


Figure 6.9: Reference types and assignment

```
static App {  
    static void Main() {  
        Size v1 = new Size();    // v1 is an instance of Size  
        Size v2 = v1;           // v2 is a 2nd instance of Size  
        v1.height = 100;  
        v2.height = 200;  
        bool truth = v2.height != v1.height;  
    }  
}
```

Figure 6.10: Using value types

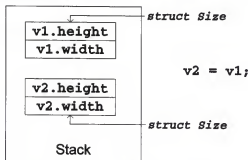


Figure 6.11: Values and assignment

Passing parameters to a method is a variation on assignment that bears special consideration. When passing parameters to a method, the method's declaration determines whether the parameters will be passed by reference or by value. Passing parameters by value (the default) results in the method/callee getting its own private copy of the parameter values. As shown in figure 6.12, if the parameter is a value type, the method gets its own private copy of the instance. If the parameter is a reference type, it is the reference that is passed by value. The object the reference points to is not copied. Rather, both the caller and callee wind up with private references to a shared object.

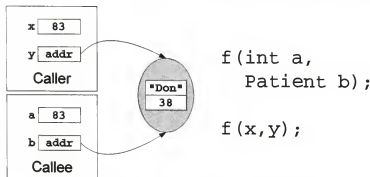


Figure 6.12: Pass by value parameters

Passing parameters by reference (indicated in C# using the `ref` or `out` modifiers) results in the method/callee getting a managed pointer back to the caller's variables. As shown in figure 6.13, any changes the method makes to the value type or the reference type will be visible to the caller. Moreover, if the method overwrites an object reference parameter to redirect it to another object in memory, this change affects the caller's variable as well.

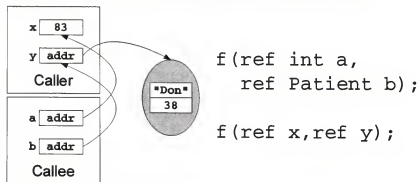


Figure 6.13: Pass by reference parameters

In the example shown in figure 6.13, any assignments the method body may perform on the `a` or `b` parameters will affect the caller. Specifically, this means that setting the `b` parameter to null will set the caller's `y` variable to null as well. In contrast, in the example shown in figure 6.12, the method body may freely assign to the parameters `a` and `b` without affecting the caller in any way. However, the object referenced by the `b` parameter is shared with the caller, and any changes made through `b` will be seen by the caller. This is true in both examples.

Cloning

Assignment of object references does not duplicate object - use cloning instead

- Cloneable objects implement the `System.ICloneable` interface
- Each type determines deep vs. shallow cloning semantics
- Built-in method `Object.MemberwiseClone` performs shallow copy
- Deep copy implemented by hand

Assigning one reference variable to another simply creates a second reference to the same object. To make a second copy of an object, some mechanism is needed to create a new instance of the same class and initialize it based on the state of the original object. The `Object.MemberwiseClone` method does exactly that, however, it is not a public method. Rather, objects that wish to support cloning typically implement the `System.ICloneable` interface, which has one method, `Clone`, as shown in figure 6.14.

```
namespace System {  
    public interface ICloneable {  
        Object Clone();  
    }  
}
```

Figure 6.14: `System.ICloneable`

The `MemberwiseClone` method performs what is called a "shallow copy," which means that it simply copies each field's value from the source object to the clone. If the field is an object reference, only the reference is copied, not the referenced object, as shown in figure 6.15. Figure 6.16 shows a class that implements `ICloneable` using shallow copies.

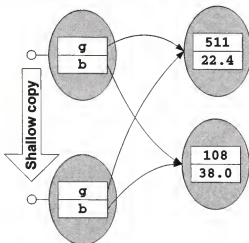


Figure 6.15: Shallow copy

```
public sealed class Person : System.ICloneable {  
    long id;  
    double age;  
    public Object Clone() {  
        return this.MemberwiseClone();  
    }  
}
```

Figure 6.16: Implementing System.ICloneable

A deep copy is one that recursively copies all objects that its fields refer to, as shown in figure 6.17. Deep copying is often what people expect, however, it is not the default behavior, nor is it a good idea to implement in the general case. In addition to causing additional memory movement and resource consumption, deep copies can be problematic when a graph of objects has cycles, since a naive recursion would wind up in an infinite loop. However, for simple object graphs, it is at least implementable, as shown in figure 6.18.

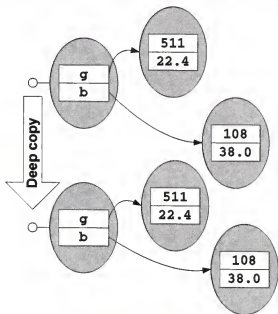


Figure 6.17: Deep copy

```
public sealed class Marriage : System.ICloneable {
    Person g;
    Person b;
    public Object Clone() {
// shallow copy first
        Marriage result = (Marriage)this.MemberwiseClone();
// deep copy each field
        result.g = (Person)(this.g.Clone());
        result.b = (Person)(this.b.Clone());
        return result;
    }
}
```

Figure 6.18: Implementing System.ICloneable

It is interesting to note that the Clone implementation in figure 6.18 could have been written without calling MemberwiseClone. An alternative implementation could have simply use the new operator to instantiate the second object and then manually populate the fields. Moreover, a private constructor could have been defined to allow the two parts (instantiation and initialization) to happen in one step. Figure 6.19 shows just such an implementation.

```
public sealed class Marriage : System.ICloneable {
    Person g;
    Person b;
    private Marriage(Person g, Person b) {
        this.g = (Person)g.Clone();
        this.b = (Person)b.Clone();
    }
    public Object Clone() {
        return new Marriage(g, b);
    }
}
```

Figure 6.19: Implementing System.ICloneable using new

Boxing

Instances of value types can be "boxed" to support object references to value types

- Value types lack "objectness" until boxed
- Boxed object is an independent clone
- Boxed object can be copied back into instance (unboxing)
- Boxing key to using System.Object as universal type

As shown in figure 6.1, all types (even value types) are compatible with `System.Object`. However, since `System.Object` is a polymorphic type, instances in memory require an object header to support dynamic method dispatching. However, value types do not have this header, nor are they necessarily allocated on the heap. To allow a value type (which ultimately is just memory) to be used in contexts that use object references such as collections or generic functions that accept `System.Object` as a method parameter, the CLR allows instances of value types to be "cloned" onto the heap in a format that is compatible with `System.Object`. This procedure is known as "boxing" and occurs whenever an instance of a value type is assigned to an object reference variable, parameter or field. For example, consider the code in figure 6.20. Note that when the instance of `Size` is assigned to an object reference variable (`itf` in this case), the CLR allocates a heap-based object that implements all of the interfaces that the underlying value type declared compatibility with. This boxed object is an independent copy, and changes to it do not propagate back to the original value type instance. However, it is possible to copy the boxed object back into a value type instance simply by using a downcast operator, as shown in 6.20. Figure 6.21 shows another example of boxing and unboxing, both visually and in code.

```
public interface IAdjustor {
    void Adjust();
}

public struct Size : IAdjustor {
    public void Adjust() { height+=2; weight+=3; }
    public int height;
    public int weight;
}

static App {
    static void Main() {
        Size s = new Size();
        bool truth = s.height == 0 && s.weight == 0;
        s.Adjust();
        truth = s.height == 2 && s.weight == 3;
        IAdjustor itf = s; // box
        itf.Adjust();      // operate on boxed copy
        truth = s.height == 2 && s.weight == 3;
        s = (Size)itf;     // unbox
        truth = s.height == 4 && s.weight == 6;
    }
}
```

Figure 6.20: Boxing

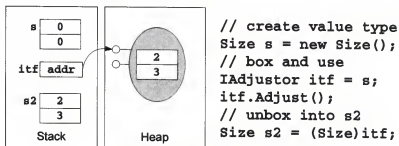
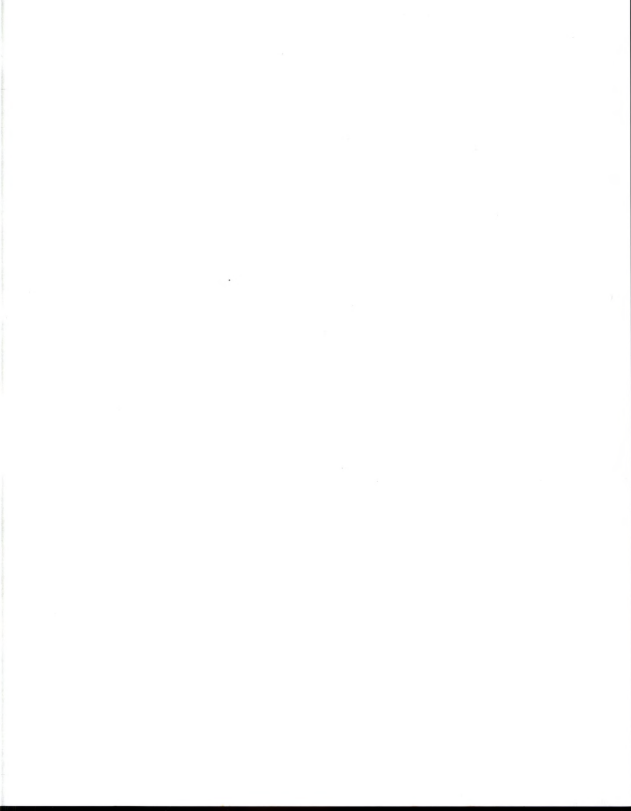


Figure 6.21: Boxing and unboxing



Object lifecycle

Objects that are no longer referenced may be garbage collected

- Static fields and "live" local variables represent root references
- Instance fields and "dead" local variables are non-roots
- GC reclaims memory for objects not reachable through at least one root reference
- GC can relocate objects in memory to compact heap
- GC typically triggered by resource thresholds
- Garbage collector exposed via static members on `System.GC` class

This chapter has focused on how objects and values are allocated and referenced. There has been no mention of how or when the underlying memory an object resides in is reclaimed over the lifetime of a running program. This is a feature. One of the primary benefits of the CLR's managed execution mode is that memory reclamation is no longer the purview of the programmer. Rather, the CLR is responsible for all memory allocation (and deallocation). The policies and mechanisms used by the CLR for managing memory are the subject of the remainder of this chapter.

The CLR tracks all object references in the system. Based on this global knowledge, the runtime can detect when an object is no longer referenced. The runtime distinguishes between "root" references and "non-root" references. A "root" reference is typically either an live local variable or a static field of a class. A "non-root" reference is typically an instance field in an object. The existence of a "root" reference is sufficient to keep the referenced object in memory. An object that has no root references is potentially no longer in use. To be exact, an object is only guaranteed to remain in memory for as long as it can be reached by traversing an object graph starting with a root reference. Objects that cannot be reached directly or indirectly via a root reference are susceptible to automatic memory reclamation (also known as garbage collection (GC)).

Figure 6.22 shows a simple object graph and both root and non-root references. Note that the set of roots is dynamic based on the execution of the program. In this example, the reachability graph shown is the one that is valid during the execution of the highlighted `ReadLine` call. Note that lexical scope is unimportant. Rather, the CLR uses liveness information created by the JIT compiler to determine which local variables are live for any given instruction pointer value.

```

public sealed class Person {
    Person spouse;
    static Person dave;
    static Person() {
        dave = new Person();
        dave.spouse = new Person();
        dave.spouse.spouse = dave;
    }
    public static void f() {
        Person temp1 = new Person();
        Person temp2 = new Person();
        System.Console.ReadLine();
        Console.WriteLine(temp1);
    }
}

```

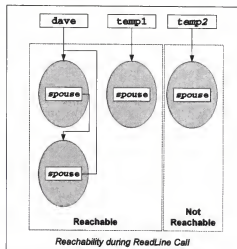


Figure 6.22: Root and non-root references

The CLR performs garbage collection when certain resource thresholds are exceeded. When this happens, the CLR takes over the CPU to track down objects that are no longer reachable via a root reference. When it finds such an object, it returns the object's memory to the heap where they can be used for another object yet to be created. The garbage collector can also relocate objects in memory to avoid heap fragmentation and to tune the process' working set by keeping live objects in fewer pages of virtual memory.

The garbage collector is exposed programmatically via the `System.GC` class. The most interesting method is `Collect`, which instructs the CLR to collect garbage now, not later. Figure 6.23 shows this method in use. Note that in this example, the object referenced by `r2` may be reclaimed at the first call to `System.GC.Collect`, as the CLR can detect that the referenced object is no longer needed, despite the fact that it is still within lexical scope in C#. By the time the second call to `System.GC.Collect` executes, the objects originally referenced by `r1` and `r3` may also be reclaimed, as `r1` is explicitly set to null and `r3` is no longer a live variable. You can trick the garbage collector into keeping an object reference "alive" by inserting a call to `System.GC.KeepAlive`. This static method does nothing other than trick the CLR into thinking that the reference passed as the parameter is actually needed, therefore keeping the referenced object from being reclaimed.

```
class UseEm {  
    static Object r1; // r1 is a root!  
    static void Main() {  
        r1 = new Object();  
        Object r2 = new Object();  
        Object r3 = new Object();  
        System.GC.Collect(); // what can be reclaimed here?  
        r1 = null;  
        r3.ToString();  
        System.GC.Collect(); // what can be reclaimed here?  
    }  
}
```

Figure 6.23: Liveness and garbage collection

The `Collect` method takes an optional parameter that controls how vast the search for unreferenced objects should be. The CLR uses a generational algorithm that recognizes that the longer an object is referenced, the less likely it is to become available for collection. The `Collect` method allows you to specify how "old" an object to consider. Be aware, however, that calling `GC.Collect` frequently can have a negative impact on performance.

Finalization

Objects can notified when they are garbage collected via finalization

- GC calls well-known `Finalize` virtual method if present
- C# destructor (`~ClassName`) implements `Finalize` and delegates to base type
- `Finalize` method always called asynchronously on a dedicated thread
- `System.GC.WaitForPendingFinalizers` waits for all pending finalizers to complete
- Garbage collection much more efficient when no finalizers are present
- Can turn finalization on/off per-object using `GC.SuppressFinalize/GC.ReregisterForFinalize`

In general, there is no need for your object to know when it is being garbage collected. All subordinate objects that your object references will themselves be automatically reclaimed as part of normal GC operation. The preferred mechanism for triggering the execution of "clean-up" code is to use a termination handler. Termination handlers protect a range of instructions inside a method by guaranteeing that a "handler" block will execute prior to leaving the protected range of instructions. This mechanism is exposed to C# programmers via the `try/finally` construct discussed in the next chapter.

Despite the existence of the termination handler mechanism, old habits often die hard, and programmers who cut their teeth on C++ often insist on tying "clean-up" code to object lifetime. To allow these old dogs to avoid learning new tricks, the CLR supports a mechanism known as object finalization. Please be aware, however, that new designs that target the CLR should avoid making use of finalization, as it is fraught with complexity and performance penalties.

Objects that wish to be notified when they are about to be returned to the heap may override the `Object.Finalize` method. When the GC tries to reclaim an object that has a finalizer, the reclamation is postponed until the finalizer can be called. Rather than reclaiming the memory, the GC enqueues the object requiring finalization onto the finalization queue. A dedicated GC thread will eventually call the object's finalizer and once the finalizer has completed execution, the object's memory is finally available for reclamation.

Your object may perform any application-specific logic in response to this notification. Be aware, however, that this method may be called long after your object has been identified by the garbage collector and will execute on one of the garbage collector's threads. A considerable amount of time can elapse between the point at which the garbage collector identifies your object as non-reachable and its finalizer being called. If your finalizer is used to release a scarce resource, in many cases it will run far later than is tolerable, which limits the utility of finalization.

Classes that override the default `Finalize` method need to call their base type's version of the method to ensure that in any base class functionality is not bypassed. In C#, you cannot implement the `Finalize` method directly. Rather, you must implement a "destructor" which causes the compiler to emit your destructor code inside a `Finalize` method followed by a call to your base type's `Finalize`. Figure 6.24 shows a simple C# class that contains a destructor. Note that the compiler-generated `Finalize` method is shown in comments.

```
public sealed class Transaction {
    int lowLevelTX;
    public Transaction() {
        lowLevelTX = raw.BeginTransaction();
    }
    public void Commit() {
        raw.CommitTransaction(lowLevelTX);
        lowLevelTX = 0;
    }
    ~Transaction() {
        if (lowLevelTX != 0)
            raw.AbortTransaction(lowLevelTX);
    }
}
/*
~Transaction is equivalent to this:
protected override void Finalize() {
    if (lowLevelTX != 0)
        raw.AbortTransaction(lowLevelTX);
    base.Finalize();
}
*/
}
```

Figure 6.24: Implementing System.Object.Finalize in C#



Programmer hygiene

Deterministic finalization is (largely) the programmer's responsibility

- Expensive resources require special attention
- The `IDisposable` interface expresses the need for deterministic/synchronous cleanup
- The `C# using` statement automates use of `IDisposable`
- Can manually suppress finalization or retrigger using `System.GC` class
- `C#'s using` protects against multiple exit points (uses `try/finally` under the hood)

Because GC is asynchronous, it is a bad idea to rely on a finalizer to clean up expensive resources. To that end, there is a standard idiom in CLR programming of providing an explicit `Dispose` method that clients can call when they are done using your object. In fact, this idiom is standardized by the `System.IDisposable` interface, which is shown in figure 6.25. Classes that implement this interface are indicating that they require explicit cleanup and that it is the client programmer's responsibility to invoke the `IDisposable.Dispose` method as soon as the referenced object is no longer needed. Since your `Dispose` method is likely to perform the same work as your `Finalize` method, it is standard practice to suppress the redundant finalization call inside of your `Dispose` method by calling `System.GC.SuppressFinalize`, as shown in figure 6.26.

```
namespace System {
    public interface IDisposable {
        void Dispose();
    }
}
```

Figure 6.25: `System.IDisposable`

```
public sealed class Transaction : IDisposable {
    int lowLevelTX;
    public int id { get { return lowLevelTX; } }
    public Transaction() {
        lowLevelTX = raw.BeginTransaction();
    }
    public void Commit() {
        raw.CommitTransaction(lowLevelTX);
        lowLevelTX = 0;
    }
    private void cleanUp() {
        if (lowLevelTX != 0)
            raw.AbortTransaction(lowLevelTX);
    }
    public void Dispose() {
        System.GC.SuppressFinalize(this);
        cleanUp();
        // call base.Dispose(); if necessary
    }
    ~Transaction() {
        cleanUp();
    }
}
```

Figure 6.26: Implementing `Dispose`

Figure 6.27 shows a client that explicitly invokes the `Dispose` method on an object once it has finished using it. The C# programming language supports the `using` statement, which tells the compiler to emit the call to `Dispose` implicitly. Figure 6.28 shows the syntax for the `using` statement. The `using` statement allows the programmer to declare one or more variables whose `IDisposable.Dispose` method will be called automatically. The syntax for the resource acquisition clause is similar to that for a local variable declaration statement. More than one variable may be declared, but the types of each of the variables must be the same. Figure 6.29 shows a simple usage of the `using` statement. Note that in this example, because the `using` statement is used with `IDisposable`-compliant objects, the compiler emits code that ensures that the `Dispose` method is invoked even in the face of unhandled exceptions or other method termination (e.g., a `return` statement).

```
class App {  
    static void Main() {  
        Transaction tx = new Transaction();  
        Console.WriteLine(tx.id);  
        tx.Dispose(); // transaction synchronously aborted  
    }  
}
```

Figure 6.27: References and deterministic finalization

Resource Acquisition/Local variable declarations

```
using (T obj1 = new T(), obj2 = new T()) {  
    // obj1 and obj2 are local variables  
} // obj2.Dispose/obj1.Dispose called here
```

Figure 6.28: The C# `using` statement

```
class App {
    static void Main() {
        using (Transaction tx = new Transaction()) {
            Console.WriteLine(tx.id);
        } // IDisposable.Dispose called automatically here
    }
}

// this code is identical to the previous using statement
Transaction tx = new Transaction();
try {
    Console.WriteLine(tx.id);
}
finally {
    if (tx != null) ((IDisposable)tx).Dispose();
}
}
```

Figure 6.29: C#'s using statement

Summary

- Objects are polymorphic and allocated on the GC heap
- Values are simply formatted memory
- Duplicate objects are created when values are boxed
- Duplicate objects are created when objects are cloned
- Objects are held in memory as long as there are "live" references to them
- Finalization (not garbage collection) has performance issues

Module 7

Delegates and Events

Delegates are runtime-aware objects that represent a method on an object or a type. Delegates are used extensively throughout the CLR as a way to glue together on a method-by-method basis. Events are simply named pseudo-properties of a type that allow delegates to be subscribe to an object or type for event notification.

After completing this module, you should be able to:

- ❑ declare and use delegates as callback routines
- ❑ declare and use multicast delegates to simulate broadcast calls
- ❑ declare and use events to simplify delegate registration
- ❑ leverage delegate-based asynchronous invocation support

Delegates and Events

Delegates are runtime-aware objects that invoke methods on other objects. Events are pseudo-properties that allow delegates to be registered against an object or type for event notification.

Motivation

Classes and interfaces are not always sufficient to model inter-component interaction

- bi-directional caller/callee relationship often desired
- Hard-coding target class type in caller not a generic solution
- Shared base class approach not flexible in world of single inheritance
- Widely used throughout the .NET framework

Figure 7.1 shows a `Worker` class that has been designed to make callbacks to a `Boss` object when certain (potentially) interesting things happen. If no `Boss` target has been registered via the `Advise` method, then the callbacks are not performed.

```
class Worker {
    public void Advise(Boss boss) { _boss = boss; }
    public void DoWork() {
        Console.WriteLine("Worker: work started");
        if( _boss != null ) _boss.WorkStarted();

        Console.WriteLine("Worker: work progressing");
        if( _boss != null ) _boss.WorkProgressing();

        Console.WriteLine("Worker: work completed");
        if( _boss != null ) {
            int grade = _boss.WorkCompleted();
            Console.WriteLine("Worker grade= " + grade);
        }
    }
    private Boss _boss;
}
```

Figure 7.1: A class-based callback relationship: caller

Figure 7.2 shows the target of the callbacks shown in the previous code fragment. Note that the implementor is only really interested in the `WorkCompleted` notification, so the other two methods are stubbed out.

```
class Boss {
    public void WorkStarted() {
        // Boss doesn't care
    }
    public void WorkProgressing() {
        // Boss doesn't care
    }
    public int WorkCompleted() {
        Console.WriteLine("It's about time!");
        return 2; // Out of 10
    }
}
```

Figure 7.2: A class-based callback relationship: target

Figure 7.3 demonstrates how these two classes might be hooked up to one another so that the `Boss` object receives callbacks from the `Worker` when certain things occur.

```
class Universe {  
    static void Main() {  
        Worker peter = new Worker();  
        Boss boss = new Boss();  
        peter.Advise(boss);  
        peter.DoWork();  
    }  
}
```

Figure 7.3: A class-based callback relationship: registration

This approach suffers from two problems. First, by using a class (`Boss`) to model the callback relationship, the `Worker` implementation is tightly coupled to the type of the callback recipient - and therefore not very generic. Secondly, any attempt to make the `Boss` class more generically useful by, for example, making each of the three notification methods `virtual`, would result in a design of limited use. This is because the CLR only allows a class to have one base class. So any interested party that wants to connect to the `Worker` object in order to receive callbacks would be required to "use up" their one chance to derived from a base class in order to be type compatible with the `Boss` base class.



Motivation (cont'd)

Interface-based designs are incrementally better than class-based designs for modeling bi-directional relationships

- More flexible than class-based design
- Does not constrain implementor's choice of base type
- Does mandate complete implementation of all methods
- As with class-based approach, requires callee to conform to/change type hierarchy

Figures 7.4, 7.5, and 7.6 attempt to address some of the issues raised in the preceding example.

```
interface IWorkerEvents {
    void WorkStarted();
    void WorkProgressing();
    int WorkCompleted();
}
```

Figure 7.4: An interface used for callbacks

```
class Worker {
    public void Advise(IWorkerEvents target) {
        _target = target;
    }
    public void DoWork() {
        Console.WriteLine("Worker: work started");
        if( _target != null ) _target.WorkStarted();

        Console.WriteLine("Worker: work progressing");
        if( _target != null ) _target.WorkProgressing();

        Console.WriteLine("Worker: work completed");
        if( _target != null ) {
            int grade = _target.WorkCompleted();
            Console.WriteLine("Worker grade= " + grade);
        }
    }
    private IWorkerEvents _target;
}
```

Figure 7.5: An interface-based callback relationship: caller

```
class Boss : IWorkerEvents {
    public void WorkStarted() {
        // Boss doesn't care
    }
    public void WorkProgressing() {
        // Boss doesn't care
    }
    public int WorkCompleted() {
        Console.WriteLine("It's about time!");
        return 2; // Out of 10
    }
}
```

Figure 7.6: An interface-based callback relationship: target

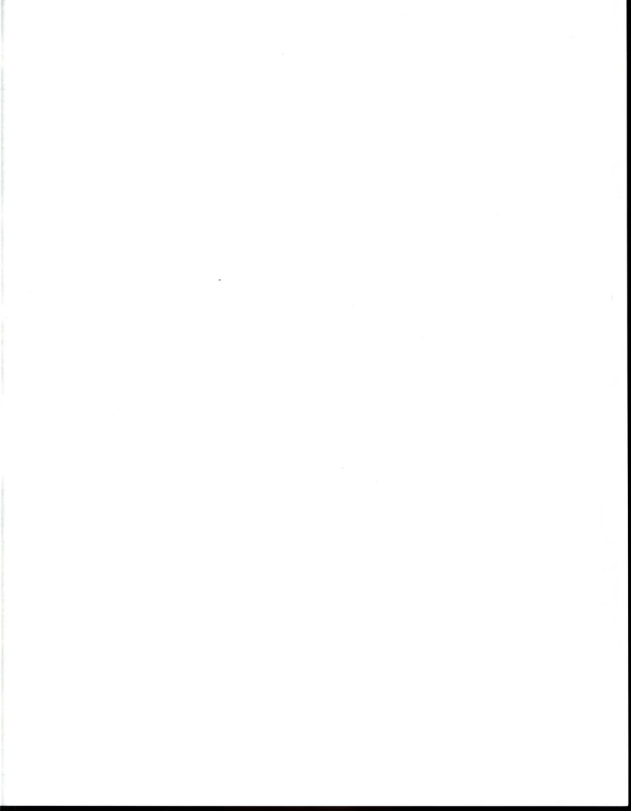
This approach uses interfaces in an attempt to make the relationship between the `Worker` class and interested parties more generic. In case, a generic `IWorkerEvents` interface is designed. Now any class that implements this interface may receive callbacks from the `Worker` class. While more generic, this approach is only incrementally better because it still suffers from a few issues.

First, in order to be compatible with the `Worker` class, the target class must modify its type hierarchy to include support for the required interface. If you are trying to connect a target object to the worker object, and you do not have access to the target component's source code, then you'd be forced to write a little shim class which implements the callback interface and forwards each method invocation on to the real target object.

Second, each method of the interface must be implemented, resulting in the same stubbed out implementation for 2 of 3 callback methods.

Third, as currently written, the `Worker` object only supports making callbacks to the most recently registered implementation of `IWorkerEvents`. If the worker object wanted to support multiple registered callback targets, extra work would be required to maintain a list of registered callback recipients.

Finally, no mechanism has been provided for a previously registered callback target to unregister from the worker object. Again - this problem is solvable, but only with extra development effort.



Delegates

A delegate is a type-safe CLR-synthesized shim that sits between a caller and zero or more call targets

- Do not require type compatibility like classes/interfaces
- Enforce only a single method signature (not a name)
- Act like a tiny interface with one method
- Facilitate component integration without source code access
- Support multiple call targets
- Support asynchronous method invocation

As shown in figure 7.7, delegate types must extend `System.MulticastDelegate` directly and must be sealed. For historical reasons, `System.MulticastDelegate` was originally factored into two classes: one that supported delegate chaining (`MulticastDelegate`) and one that did not (`Delegate`). As of July 2001, all delegates are required to derive from `System.MulticastDelegate`, rendering the `System.Delegate` type a historical anomaly.

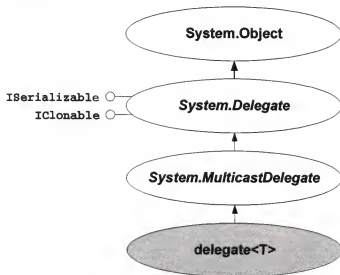


Figure 7.7: Delegate type hierarchy

Declaring Delegates

Language-specific syntax is used to define delegates

- C#/VB.NET use function prototype syntax
- Compiler synthesizes user-defined type
- User-defined type extends `System.MulticastDelegate`

Figure 7.8 demonstrates how a delegate is declared in C#. In this example, three delegates are declared - one for each type of notification the worker object would like to support. Note that since the `WorkStarted` and `WorkProgressing` delegates represent a delegate that's capable of invoking any method that takes and returns no arguments, only one of these delegate types would be necessary in practice.

```
delegate void WorkStarted();
delegate void WorkProgressing();
delegate int WorkCompleted();
```

Figure 7.8: Declaring user-defined delegate

Figure 7.9 illustrates the language mapping from a C# delegate declaration onto the corresponding CLR type system. This diagram is actually only a partial listing of what the compiler produces. The remainder of the output (dealing with asynchronous invocation) has been omitted here and will be revisited later in this module. For now, note that delegate types have a constructor that takes a reference to the target object and an indication of which method on that target object the delegate should be bound to. Also note that an `Invoke` method is declared that has the identical parameter list and return value as was declared in the delegate type declaration.

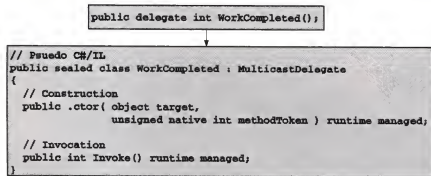


Figure 7.9: C# to CLR language mapping

Using Delegates

Language-specific syntax is used to instantiate and invoke delegates

- C#/VB.NET use constructor syntax to bind delegate instance to target method
- C#/VB.NET use method invocation syntax to invoke target method
- Compiler maps invocation syntax onto `MulticastDelegate.Invoke`
- Client passes delegate to caller for future use

Figure 7.10 shows how the worker class would be rewritten to make calls through previously registered delegates. Note that standard C# function call syntax is used to invoke a delegate synchronously. In this example, the delegate types were used to declare three public fields that clients could access in order to register themselves with the worker object (by assignment in this case, instead of a public `Advise` method).

```
class Worker {
    public void DoWork() {
        Console.WriteLine("Worker: work started");
        if( started != null ) started();

        Console.WriteLine("Worker: work progressing");
        if( progressing != null ) progressing();

        Console.WriteLine("Worker: work completed");
        if( completed != null ) {
            int grade = completed();
            Console.WriteLine("Worker grade= " + grade);
        }
    }

    public WorkStarted started;
    public WorkProgressing progressing;
    public WorkCompleted completed;
}
```

Figure 7.10: Calling through a delegate

Figure 7.11 shows a revised `Boss` class. Note that since this type is only interested in registering for notifications about the `WorkCompleted` event, it only needs to implement one method.

```
class Boss {
    public int WorkCompleted() {
        Console.WriteLine("Better...");
        return 4; // Out of 10
    }
}
```

Figure 7.11: Simplified target

Figure 7.12 shows how the `Boss` object's target method is passed to the constructor for the `WorkCompleted` delegate. This newly initialized delegate is then registered with the worker object by setting the worker object's `completed` field.

```
class Universe {  
    static void Main() {  
        Worker peter = new Worker();  
        Boss boss = new Boss();  
        peter.completed = new  
WorkCompleted(boss.WorkCompleted);  
        peter.DoWork();  
    }  
}
```

Figure 7.12: Delegate Initialization

Figure 7.13 illustrates what the previous bits of sample code have accomplished at run-time.

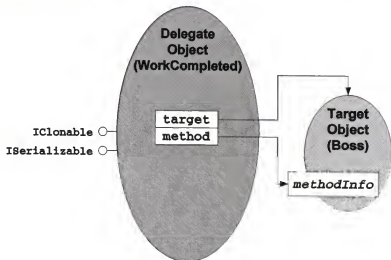


Figure 7.13: A delegate instance

It's important to note that delegates allow components to be integrated without changing the type hierarchy of the callback target. This means that you could connect a target object to something like the `Worker` class even without access to the target object's source code. As long as the target object contains a method that matches the particular delegate signature you're interested in, a delegate can be instantiated and bound to that method, and then registered with the calling piece of code (the worker in this example).



Supporting Multiple Targets

Delegates support multiple target registration and invocation

- Language-specific [un]registration operators provided by C#/VB.NET
- Multiple targets supported automatically
- Registered targets called *sequentially* by *Invoke*
- Manual iteration of registered targets supported, but optional
- Both instance and static method targets supported

Delegates provide more than just an alternative syntax for designing callback relationships. In addition to alleviating the forced type compatibility of the target type, delegates intrinsically support the registration, unregistration, and invocation of multiple registered targets. Delegates may also be bound to both instance and static methods, and subsequently treated the same by the caller.

Figure 7.14 demonstrates the use of the C# += operator to register additional targets with the `completed` field of the `worker` class. This is the preferred mechanism for registering delegates, since it does not overwrite or replace any previously registered targets. Using the simple assignment operator, while still supported at this point in the story, is not very polite.

```
class Universe {
    static void WorkerStartedWork() {
        Console.WriteLine("Universe notices working starting");
    }
    static int WorkerDoneWorking() {
        Console.WriteLine("Universe notices is worker done");
        return 7;
    }
    static void Main() {
        Worker peter = new Worker();
        Boss boss = new Boss();

        // Impolite registration syntax:
        peter.completed = new
WorkCompleted(boss.WorkCompleted);
        peter.started = new
WorkStarted(Universe.WorkerStartedWork);
        peter.progressing = null; // Intended?

        peter.completed(); // Intended?

        // Preferred registration syntax:
        peter.completed += new
WorkCompleted(WorkerDoneWorking);
        peter.DoWork();
    }
}
```

Figure 7.14: Multiple target registration

When multiple delegates are registered in this fashion, the calling code does not need to do anything special to invoke all of the registered targets. The exact same function calling syntax will cause the `Invoke` method to iterate through each registered target, calling each sequentially (not in parallel). However, if the caller would like to manually call each target, the `GetInvocationList` method may be used to fetch an array containing the

set of registered targets. In our example, this technique might be used to allow the return value of each delegate invocation to be harvested and displayed. This approach is demonstrated in Figure 7.15.

```
class Worker {
    public void DoWork() {
        // Start and progress with work...

        Console.WriteLine("Worker: work completed");
        if( completed != null ) {
            foreach( WorkCompleted wc in
completed.GetInvocationList() ) {
                int grade = wc();
                Console.WriteLine("Worker grade= " + grade);
            }
        }
    }

    public WorkStarted started;
    public WorkProgressing progressing;
    public WorkCompleted completed;
}
```

Figure 7.15: Iterating through registered targets

Figure 7.16 illustrates what the system would look like if multiple targets were registered with the `completed` delegate field.

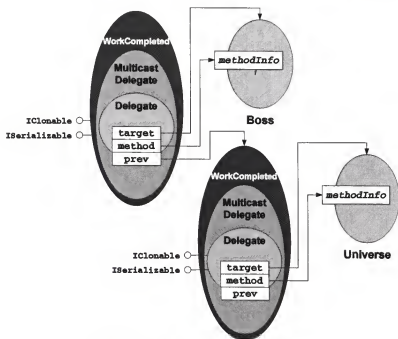


Figure 7.16: A multicast delegate

Events

Events are the formalized use of delegates that support public [un]registration and private invocation

- Using public fields for delegate registration offers too much access
- Client can overwrite previously registered target(s)
- Client can invoke target(s)
- Burden placed on client to be "polite"
- Public registration methods coupled with private delegate field is better, but tedious if done manually
- `event` modifier automates support for public [un]registration and private implementation

At this point, we've shown that delegates provide a non-trivial amount of built-in support for features such as supporting multiple targets. However, as alluded to in Figure 7.14, the burden is on clients to make polite use of delegates that have been used the way the `Worker` has used them. Specifically, clients must voluntarily avoid the use of assignment and instead use the `+=` and `-=` operators for registering and unregistering delegates in order to avoid overwriting previously registered delegates. Similarly, the client should avoid invoking the delegate themselves - something the `Worker` class author probably does not intend to support. The event modifier allows the class author to formalize the "appropriate" or intended use of such delegates by instructing the compiler to only allow registration and unregistration of delegates, while disallowing assignment and invocation.

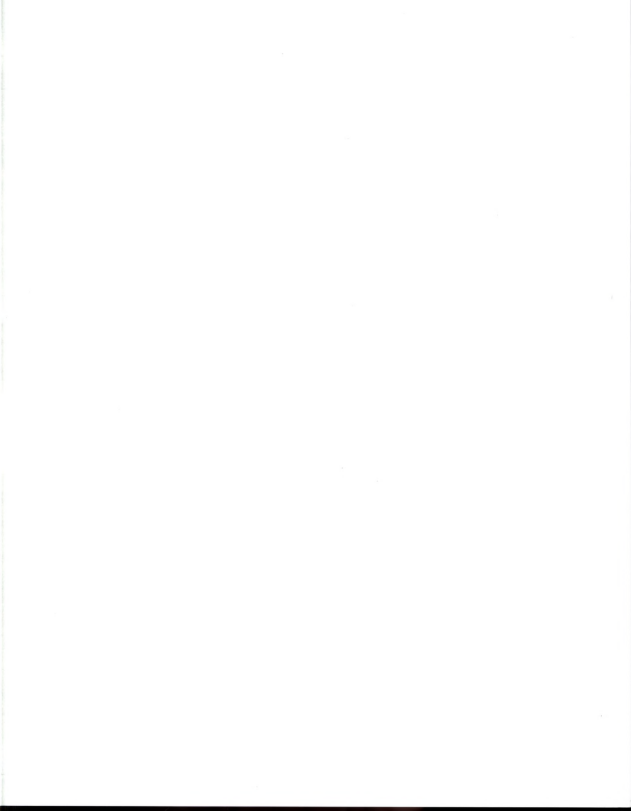
Figure 7.17 shows a revised `Worker` class that uses the event modifier. Figure 7.18 revisits the previous "hookup" code, with source code annotations indicating which operations would now cause compilation errors.

```
class Worker {  
    // ...  
    public event WorkStarted started;  
    public event WorkProgressing progressing;  
    public event WorkCompleted completed;  
}
```

Figure 7.17: Declaring events

```
class Universe {  
    // ...  
    static void Main() {  
        Worker peter = new Worker();  
        Boss boss = new Boss();  
  
        // Illegal: assignment to event field not supported:  
        peter.completed = new  
        WorkCompleted(boss.WorkCompleted);  
        peter.started = new  
        WorkStarted(Universe.WorkerStartedWork);  
        peter.progressing = null;  
  
        // Illegal: execution of event not supported:  
        peter.completed();  
  
        // Registration still supported:  
        peter.completed += new  
        WorkCompleted(WorkerDoneWorking);  
        peter.DoWork();  
    }  
}
```

Figure 7.18: Event registration



Customizing Event Registration

User-defined event registration handlers may optionally be provided

- The one benefit of writing your own registration methods is control
- Alternative property-like syntax supports user-defined registration handlers
- Allows you to make registration conditional or otherwise customized
- Client-side access syntax not affected
- You must provide storage for registered clients

The one benefit of providing your own Advise and Unadvise methods for [un]registration of delegates is that it allowed you to encode or enforce rules or error handling around those two activities. For example, you might want to only support a fixed number of simultaneously registered callback clients. Or maybe you want to make registration condition on some criteria - like the identity of the caller, the number of available resources at your disposable, or any other criteria. For these scenarios, the C# language allows you to specify custom registration and unregistration handlers using syntax very similar to property declarations. Instead of using `get` and `set` blocks to implement the getter and setter functionality for a property, you instead use `add` and `remove` blocks to implement the registration and unregistration methods.

7.19 demonstrates custom event registration handler syntax. In this example, the registration handler for the `progressing` event only allows clients to register if it's not yet noon. Note that once you leverage this syntax, you are required to maintain the actual list of registered delegates yourself.

```
class Worker {  
    public void DoWork() { ... }  
  
    public event WorkStarted started;  
    public event WorkCompleted completed;  
  
    public event WorkProgressing progressing {  
        add {  
            if( DateTime.Now.Hour < 12 ) {  
                _progressing += value;  
            }  
            else {  
                string msg = "Must register before noon.";  
                throw new InvalidOperationException(msg);  
            }  
        }  
        remove {  
            _progressing -= value;  
        }  
    }  
  
    private WorkProgressing _progressing;  
}
```

Figure 7.19: Customizing event [un]registration

Asynchronous Invocation

Delegates are the preferred approach to asynchronous method invocation in the CLR

- Compiler/CLR-synthesized methods support non-blocking method invocation
- `BeginInvoke` queues request-to-call-delegate to CLR-managed thread pool
- `IAsyncResult` returned for optional completion discovery
- `EndInvoke` harvests target method return value and outputs (including exceptions thrown by target)
- May fire-and-forget, poll for completion, or block voluntarily
- May request callback when target invocation completes
- Beware thread synchronization issues within callbacks

Figure 7.20 shows the complete language mapping for a delegate named `Add` that takes 4 parameters and returns a value. Note that in addition to the constructor and synchronous invocation support (`Invoke`), two other methods are also defined. `BeginInvoke` is used to enqueue a request to call the target with a CLR-managed pool of threads. At some point in the future (as soon as possible) one thread from this pool will be dispatched to invoke the requested target method. The return value of `BeginInvoke` represents a reference to an object that tracks the state of the requested method call (expressed by the `IAsyncResult` interface). This interface can be used to detect when the target method has been invoked, and what the results (if any) were. This method may also be used to request a callback to be invoked once the target method invocation has been completed.

```
public delegate double Add( double w, double x,
                           ref double y, out double z );
```

```
// Pseudo C#/IL
public sealed class Add : MulticastDelegate
{
    // Construction
    public .ctor( object target,
                 unsigned native int methodToken ) runtime managed;

    // Invocation
    public double Invoke( double w, double x,
                        ref double y, out double z ) runtime managed;

    public IAsyncResult BeginInvoke( double w, double x,
                                    ref double y, out double z,
                                    AsyncCallback cb,
                                    object state ) runtime managed;

    public double EndInvoke( ref double y, out double z,
                           IAsyncResult ar ) runtime managed;
}
```

Figure 7.20: C# to CLR language mapping - asynchronous invocation

Figure 7.21 provides an example meant to illustrate one reason why asynchronous method invocation might be desired.

```
class Boss {
    public int WorkCompleted() {
        System.Threading.Thread.Sleep(3000);
        Console.WriteLine("Better...");
        return 5; // Out of 10
    }
}

class Universe {
    static int WorkerDoneWorking() {
        System.Threading.Thread.Sleep(4000);
        Console.WriteLine("Universe notices is worker done");
        return 8;
    }
    ...
}
```

Figure 7.21: Dealing with lengthy operations

Figure 7.22 shows one approach to leveraging asynchronous method invocation. In this example, the caller simply fires off the request to invoke the target, and forgets about everything to do with this method request - moving on to the next registered target. In situation, the caller does not know when any of the target methods have been invoked, nor what their outcome might have been. Figure 7.23 illustrates this situation at run-time.

```
class Worker {
    public void DoWork() {
        // Start and progress with work...

        Console.WriteLine("Worker: work completed");
        if( completed != null ) {
            foreach( WorkCompleted wc in
                completed.GetInvocationList() ) {
                wc.BeginInvoke(null, null); // Fire-and-forget
            }
        }
    }

    public event WorkProgressing progressing;
}
```

Figure 7.22: Async invocation - fire-and-forget

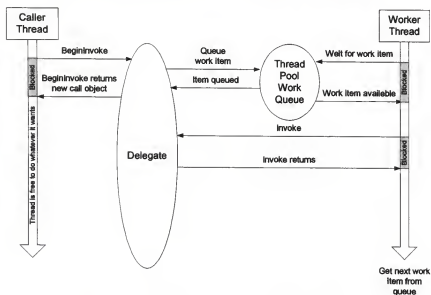


Figure 7.23: Async delegates - fire-and-forget

Figure 7.24 demonstrates the use of the `AsyncResult.IsCompleted` property to poll for the completion of the target method invocation. Note that once the method has completed, the `EndInvoke` method is called to harvest the return value and output parameters (if any) of the target method. These results would have been stored in the call object illustrated on figure 7.25. Note that if the target had raised an exception, the reference to the exception object would also have been stored in the call object just as with the regular return value. In this situation, the call to `EndInvoke` will trigger the re-raising of the exception - at which point the caller would use the standard mechanisms provided by their language for dealing with exceptions.

```

class Worker {
    public void DoWork() {
        // Start and progress with work...

        Console.WriteLine("Worker: work completed");
        if( completed != null ) {
            foreach( WorkCompleted wc in
completed.GetInvocationList() ) {
                IAsyncResult ar = wc.BeginInvoke(null, null);
                while( !ar.IsCompleted ) {
                    System.Threading.Thread.Sleep(1000); // or be
more useful
                }
                int grade = wc.EndInvoke(ar);
                Console.WriteLine("Worker grade= " + grade);
            }
        }
    }
}

public event WorkProgressing progressing;
}

```

Figure 7.24: Async invocation - polling for completion

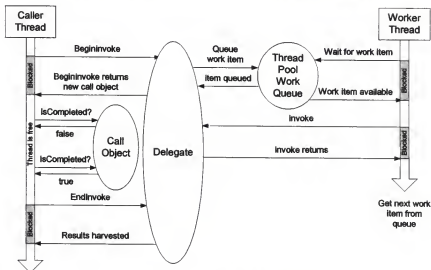


Figure 7.25: Async delegates - polling

Figure 7.26 shows one alternative to polling, which is to block voluntarily on the `AsyncWaitHandle` property of the `IAsyncResult` interface. This

approach has two advantages over polling. First, it avoids wasting processor resources until the target method invocation has been completed. Second, it provides a convenient way to place a timeout on the wait operation.

```
class Worker {
    public void DoWork() {
        // Start and progress with work...

        Console.WriteLine("Worker: work completed");
        if( completed != null ) {
            foreach( WorkCompleted wc in
completed.GetInvocationList() ) {
                IAsyncResult ar = wc.BeginInvoke(null, null);
                if( ar.AsyncWaitHandle.WaitOne(250, true) )
                {
                    int grade = wc.EndInvoke(ar);
                    Console.WriteLine("Worker grade= " + grade);
                }
                // else, timed out waiting for result
            }
        }
    }
}

public event WorkProgressing progressing;
}
```

Figure 7.26: Async invocation - blocking

The third approach to discovering the completion of an asynchronous method invocation request is to include a delegate of type `AsyncCallback` with your request to the thread pool. Once the target delegate has been invoked, the thread from the thread pool that serviced that request will then make a call back to you, informing you that the invocation has been completed. This approach, and the resulting run-time depiction, is illustrated in Figures 7.27 and 7.28.

```

class Worker {
    public void DoWork() {
        ...
        Console.WriteLine("Worker: work completed");
        if( completed != null ) {
            foreach( WorkCompleted wc in
completed.GetInvocationList() ) {
                AsyncCallback myCallback = new
AsyncCallback(WorkGraded);
                wc.BeginInvoke(myCallback, wc);
            }
        }
    }

    private void WorkGraded( IAsyncResult ar ) {
        WorkCompleted wc = (WorkCompleted)ar.AsyncState;
        int grade = wc.EndInvoke(ar);
        Console.WriteLine("Worker grade= " + grade);
    }

    public event WorkProgressing progressing;
}

```

Figure 7.27: Async invocation - completion callback

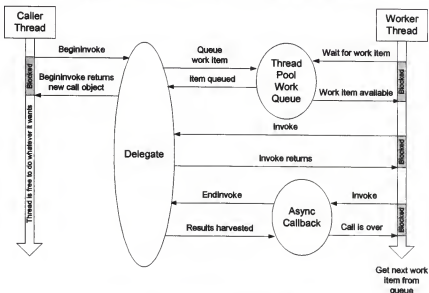
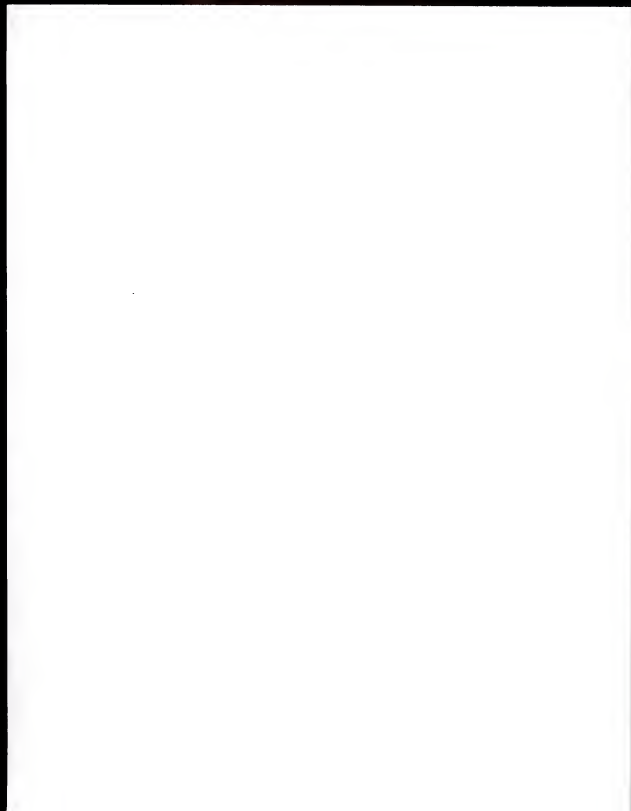


Figure 7.28: Async delegates - callback

Summary

- Delegates are objects that invoke methods on other objects
- Delegates are instances of delegate types
- `System.Delegate` is a "magic" type to the runtime
- Asynchronous invocation is possible using `System.Delegate`
- Events are registration/deregistration methods with well-known semantics



Module 8

Threads

The CLR provides a multithreaded execution environment. By default, objects can be accessed simultaneously by multiple threads. This module explains how threads work in .NET, and the services that are available to protect your objects from multithreaded access.

After completing this module, you should be able to:

- understand how and when Threads are created
- understand what mechanisms are provided for synchronizing threads

Threads

Threads permeate most .NET applications as a result of threading, delegates, and other activates. The CLR provides several tools for creating and managing threads.

Thread Origins

All but exceedingly trivial applications have multiple threads in them.

- A thread is an independent path of execution through code that is scheduled for time on the processor in a preemptive fashion.
- Several activities introduce threads into an application
- Calling `BeginInvoke` against a delegate
- Using `System.Threading.ThreadPool` class to request a callback ASAP
- Using `System.Threading.Timer` class to schedule a callback
- Manually creating and starting a new thread
- Every scenario provides an opportunity for timing-related errors

Calling `BeginInvoke` against a delegate causes the method invocation request to be "packaged up" and posted to a shared pool of threads. As soon as one of the pooled threads is available, it will perform the actual function call against the target method on your behalf, caching output results (including the return value) for later retrieval when you call `EndInvoke`. If you request a callback when you call `BeginInvoke` (as shown in figure 8.1), that callback is made by a thread from the thread pool. If your callback accesses data that the main threads in your application can also access, you have a thread synchronization issue to worry about.

```
using System;

public class App
{
    delegate void SimpleFuncDelegate();

    public static void Main() {
        SomeFunction(42);
        SimpleFuncDelegate fxn =
            new SimpleFuncDelegate(SomeFunction);
        fxn.BeginInvoke( 42,
            new AsyncCallback(OnComplete),
            null );
        // Go do other things...
    }
    static void SomeFunction( int x ) {
        // What thread is this called on?
    }
    static void OnComplete( IAsyncResult ar ) {
        // What thread is this called on?
    }
}
```

Figure 8.1: Threading using delegates

A pool of reusable threads is maintained for each process. Async delegate invocation is the most common way to use those threads. However, you can explicitly request that a thread from that thread pool call you back by passing an instance of the system-supplied `WaitCallback` delegate to the `QueueUserWorkItem` method of the `ThreadPool` class. Figure 8.2 demonstrates this technique. As with `BeginInvoke`, the next available thread from the thread pool will call you back at the specified location.

do intervals change based on how many threads you have?

```
using System;
using System.Threading;

public class App
{
    public static void Main()
    {
        ThreadPool.QueueUserWorkItem(
            new WaitCallback(SomeFunction), 42);
        // Go do other things...
    }
    static void SomeFunction( object arg )
    {
        // What thread is this called on?
        int x = (int)arg;
    }
}
```

Figure 8.2: Threading using the ThreadPool class

The CLR also provides a `Timer` class, which allows you to schedule a callback from a thread in the thread pool at a particular time in the future, expressed in terms of elapsed time from now. Furthermore, you can specify whether or not the callback continues to be rescheduled automatically at some fixed interval (a periodic timer), or if the timer should be a one-shot timer that automatically expires after the first callback is made at the scheduled time. Figure 8.3 demonstrates the use of the timer class.


```
using System;
using System.Threading;

public class App
{
    public static void Main()
    {
        // Fire 5 seconds from now,
        // repeating at 1 second intervals.
        int TicksPerSec = TimeSpan.TicksPerSecond;
        new Timer( new TimerCallback(OnTimerFired),
            42,
            new TimeSpan(TicksPerSec*5),
            new TimeSpan(TicksPerSec*1) );
        // Go do other things...
    }
    static void OnTimerFired( object arg ) {
        // What thread is this called on?
        int x = (int)arg;
    }
}
```

Figure 8.3: Threading using timers

Using any of the above 3 techniques to temporarily "borrow" the use of a thread from the CLR-maintained thread pool typically provides the most convenient and efficient form of multithreading. But in each of those scenarios, the callback you implement should be aware of the fact that you're using a borrowed thread - and return as fast as possible from your callback. If you want to dedicate a thread to a long-running activity (maybe a background activity that you perform during the life of your application), or if you want to have the ability to prioritize different activities in your application, then the CLR provides a `Thread` class that allows you to explicitly create a new thread of execution. You can then start, stop, suspend/resume, and prioritize this new thread as you see fit. Figure 8.4 demonstrates how to start a new dedicated thread of execution.

Module 8: Threads

A thread started manually is a foreground thread, thread pool a background threads

```
using System;
using System.Threading;

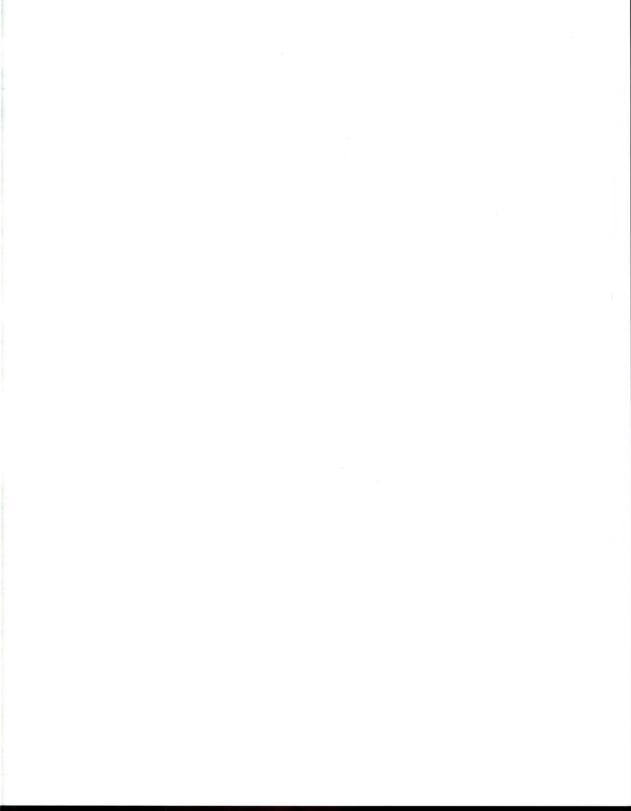
public class App
{
    public static void Main()
    {
        Thread t = new Thread(new ThreadStart(MyIdleThread));
        t.Priority = ThreadPriority.Lowest;
        t.Start();

        // Go do other things...

        t.Interrupt(); // Stop the thread.
        t.Join(); // Wait for thread to exit.
    }
    static void MyIdleThread() {
        try {
            // Do long-running activity, low priority,
            // activity here...
        }
        finally {
            // Do cleanup here.
        }
    }
}
```

Figure 8.4: Threading using the Thread class

You must consider things like performance and scalability, security, call context, and prioritization when choosing among these different modes of multithreading. But in all 4 scenarios, you must be aware that you now have the potential for timing-related issues that result when multiple threads try to access the same data.



Threading Comparisons

All threading scenarios cause code to execute on a separate CLR thread, but they differ in terms of performance, scalability, and context-awareness.

- Use of shared thread pool scales to large work loads
- CAS markers on initiating thread stack always propagated
- Security principal of initiating thread not always propagated
- Call context of initiating thread not always propagated

Any technique that leverages the built-in thread pool will scale well if a very large number of requests are placed against the pool. In other words, making 1000 requests against the thread pool using `BeginInvoke` will not cause the thread pool manager to create (or attempt to create) 1000 threads to handle those requests. Creating your own thread is more useful where you want to dedicate a thread to a particular long-running activity, possibly prioritizing that thread relative to other work being done by your program.

All techniques share one thing in common: they introduce a new thread of execution into your program. This means that, no matter which technique you use for a particular task, you will need to consider the ramifications of having multiple threads simultaneously access shared state.

Beyond just the ability to perform work on a separately scheduled thread of execution, each of the techniques discussed so far vary slightly in terms of how they deal with any CAS markers current in place on the initiating thread's callstack, whether or not the initiating thread's current security principal is propagated to the target thread, and whether or not information currently stored in call context for the initiating thread is available in the target thread. Figure 8.5 summarizes these differences. These differences should be taken into account when considering which asynchronous execution technique to leverage for a particular task.

	Scalable?	CAS Markers Flow?	Thread Principal Flows?	Call Context Flows?
<code>Delegate.BeginInvoke</code>	yes	yes	yes	yes
<code>ThreadPool.QueueUserWorkItem</code>	yes	yes	no	no
<code>new Timer</code>	yes	yes	no	no
<code>new Thread</code>	no	yes	yes	no

Figure 8.5: Comparing asynchronous execution alternatives

Thread Synchronization

Thread synchronization refers to the act of defending against multithreading issues such as races, deadlock, and starvation.

- A per-object SyncBlock is the primary defense
- Each object has a demand-allocated SyncBlock
- Operated on using static methods of the `Monitor` class
- Simple use of `Enter` and `Exit` methods provides in single-threaded access
- C# and VB.NET provide exception-safe wrappers (`lock`, `SyncLock`)
- Can be used on `Type` objects to protected statics
- Can only grab SyncBlock of objects in same `AppDomain`

By default, the runtime allows multiple threads to access objects simultaneously. Unless the programmer requests the runtime to do otherwise, either by using an attribute-based approach or by explicit programming operations, concurrent access to objects is allowed. If multithreaded access to a particular object could create problems, the runtime provides several different ways to ensure the correct operation of your object in the presence of multiple threads.

Each instance of a reference type can have a single `SyncBlock`, which is referred to by a `SyncBlockIndex` property. If no synchronization operations have been performed on an object, the `SyncBlockIndex` (which is an implicit field of each object) is 0. If `Object.GetHashCode` is called on an object, the runtime will allocate the `SyncBlockIndex`, but not an actual `SyncBlock`. This saves memory. Only when an actual synchronization operation, like calling `Monitor.Enter` or using the C# `lock` construct will cause the `SyncBlock` for an object to be allocated. Figure 8.6 demonstrates the use of `Monitor.Enter` and `Exit`, along with the corresponding C# `lock` construct. Figure 8.7 demonstrates the user of the monitor to protect access to statics.

```
public class Foo
{
    public void ThreadSafeOperation() {
        Monitor.Enter(this);
        // Do something that requires single-threaded access
        Monitor.Exit(this);
    }
    public void JustAsThreadSafeOperation() {
        lock(this)
        {
            // Do something that requires single-threaded access
        }
    }
}
```

Figure 8.6: Using a Monitor

```
public class Foo
{
    private static int PrevValue = 0;
    private static int NextValue = 1;

    public void ThreadSafeOperation()
    {
        // Use the syncblock for the Foo type object
        // to protect type variables (statics).
        //
        lock( typeof(Foo) )
        {
            PrevValue = NextValue;
            NextValue++;
        }
    }
}
```

Figure 8.7: Protecting Statics

The C# `lock` and VB.NET `SyncBlock` constructs provide an exception-safe wrapper around calls to `Monitor.Enter(this)` and `Monitor.Exit(this)`. For example, the following use of the C# `lock` construct:

```
public void FooMethod()
{
    lock(this)
    {
        // Do protected work
    }
}
```

is equivalent to the following use of the `Monitor` class:

```
public void FooMethod()
{
    Monitor.Enter(this);
    try
    {
        // Do protected work
    }
    finally
    {
        Monitor.Exit(this);
    }
}
```




Denial of Service

Unsafe, incorrect, or malicious use of `Monitor.Enter` could cause problems for objects synchronizing on their own object's syncblock

- External agent synchronizes on an object using `Monitor.Enter(refToYourObject)`
- An exception, errant code path, or malicious intent causes `Monitor.Exit` to be missed
- Subsequent use of `lock(this)` from within object now blocks
- Defense: synchronize on a private member
- Cost of defense: extra objects

Figure 8.8 shows the use of a private internal member variable for self-synchronization.

```
public class Foo
{
    private object syncObj = new object();

    public void Method1()
    {
        lock(syncObj)
        {
            // ...
        }
    }

    public void Method2()
    {
        lock(syncObj)
        {
            // ...
        }
    }
}
```

Figure 8.8: Defending against external SyncBlock acquisition

Fairness

Simple synchronization primitives can eliminate nondeterministic outcomes, but often offer no protection against indefinite postponement

- Starvation can happen when multiple threads must contend for a resource in a thread-safe manner, but block if no resource is available
- Good: grab lock, if resource available consume, release lock
- Bad: grab lock, if resource not available wait, release lock
- Must release lock before waiting - then reacquire once resource becomes available
- With simple locks, releasing then reacquiring puts you in the back of the line, which could happen indefinitely (starvation)
- Monitors are designed to prevent this problem

A Monitor can be used to provide higher levels of serialized access, above and beyond simple mutual exclusion. As an example, imagine a thread-safe queue class that has a `Dequeue` operation that will block the caller until an element has been placed on the queue for consumption. On entry into the `Dequeue` method, `Monitor.Enter` is called to ensure that only one thread at a time enters the monitor (and therefore any methods on the object). However, if after acquiring the lock it is determined that there are no entries in the queue, the calling thread needs to be blocked. If this blocking operation is done while holding the lock, deadlock will ensue. Releasing the lock before waiting will prevent deadlock, but might result in starvation. This is because another thread could enter the monitor while the first thread is sleeping and discover that an entry has subsequently been placed in the queue; dequeuing it for consumption. This could happen repeatedly, causing the original thread to wait indefinitely for his shot at dequeuing an element.

The support this type of design in a fair manner, the `Monitor` provides additional `Wait` and `Pulse` operations. These methods can only be called within the scope of an `Enter/Exit` pair. Calling `Wait` causes the monitor lock to be released and the calling thread blocked. This would be done by the `Dequeue` method. When a resource subsequently becomes available, the `Enqueue` method could call `Pulse` to tell the monitor to release one of the threads currently blocked in a call to `Wait`. The runtime will ensure that the thread that returns from `Wait` in this manner will have reacquired the monitor lock. The `Dequeue` method still calls `Exit` to leave the monitor, but preferential treatment has already been given to threads blocked in `Wait`, so starvation is avoided. Figure 8.9 demonstrates a thread safe queue class that uses `Pulse` and `Wait` to ensure fairness.

```
public class Queue
{
    public object Dequeue() {
        object element = null;

        Monitor.Enter(this);
        if( InternalList.IsEmpty() )
        {
            Monitor.Wait(this);
        }
        element = InternalList.RemoveFromHead();
        Monitor.Exit(this);

        return(element);
    }
    public void Enqueue( object element ) {
        Monitor.Enter(this);
        InternalList.AddToTail(element);
        Monitor.Pulse(this);
        Monitor.Exit(this);
    }
}
```

Figure 8.9: Using a Monitor to Prevent Starvation



Read/Write-Aware Synchronization

Simple mutual exclusion can be "too safe" when threads are only trying to read data

- Any number of threads can safely read data concurrently
- Only when threads are updating data is locking needed
- Reader threads can acquire lock if and only if 0 writers
- Writer threads can acquire lock if and only if 0 readers and 0 writers
- The `ReaderWriterLock` supports read/write-aware synchronization
- Only the programmer can classify "read" versus "write" activities

It's up to the programmer to identify places where concurrent access to shared data by multiple is "safe" (i.e., the reading activities), and where concurrent access is not safe (the writing activities). Once you've identified those locations in your code, you make corresponding calls to methods on an instance of the `ReaderWriterLock` class. The `ReaderWriterLock` will take care of enforcing the multiple readers/single writer policy from that point on. Figure 8.10 demonstrates the use of the `ReaderWriterLock` class.

```
using System.Threading;

public class Foo
{
    private ReaderWriterLock rwLock;
    private int x;
    private int y;

    public Foo() {
        rwLock = new ReaderWriterLock();
    }

    public void GetInts( ref int a, ref int b )
    {
        rwLock.AcquireReaderLock(Timeout.Infinite);
        a = x; b = y;
        rwLock.ReleaseReaderLock();
    }

    public void SetInts( int a, int b )
    {
        rwLock.AcquireWriterLock(Timeout.Infinite);
        x = a; y = b;
        rwLock.ReleaseWriterLock();
    }
}
```

Figure 8.10: Using `ReaderWriterLock`

Synchronization Primitives

In addition to object-based, the CLR provides a full suite of classic manual synchronization techniques

- **Interlocked** class provides atomic, non-blocking integer updates
- **Mutex** class provides single-threaded access
- **(Auto,ManualReset)Event** classes provide notification primitives
- **Wait-based** techniques use abstraction of a **WaitHandle** class for perform waits on one or more waitable objects
- **Manual** techniques offer two advantages over **SyncBlock**-based techniques:
(1) can be used for interprocess synchronization and (2) deadlock-free waits for multiple resources

The `Interlocked` class provides static methods for performing thread-safe updates to integer data elements. These operations use processor-specific instructions to perform atomic (uninterruptible) read/modify/write operations on individual integers. No blocking is ever performed, making these operations several orders of magnitude faster than any of the wait-based operations. If the integer(s) being operated on happen to be in shared memory that's being used by more than one process (using `P/Invoke` to call `CreateFileMapping` and `MapViewOfFile`), then these operations can be performed on integers that reside in shared memory using C# pointer operations from within `unsafe` methods. Figure 8.11 demonstrates the use of the `Interlocked` class.

```
public class Cat
{
    private int CloseCalls = 0;
    private int LivesLeft = 0;

    public void OnFallFromTree()
    {
        Interlocked.Increment(ref CloseCalls);
        Interlocked.Decrement(ref LivesLeft);
    }
}
```

Figure 8.11: Using the `Interlocked` class

The `mutex` class provides for mutually exclusive (i.e., single threaded) access to shared resources - just like the `Monitor` class and `SyncBlock`. However, the `mutex` class allows the `mutex` object to be assigned a name, which then allows the same `mutex` to be opened and used from another process on the same machine. The following example demonstrates the use of the `Mutex` class:

```
public class Bank {
    private Mutex accountLock = new Mutex();
    public Funds Withdraw( int HowMuch ) {
        Funds money = null;

        accountLock.WaitOne();
        money = DebitFromThisAccount(HowMuch);
        accountLock.ReleaseMutex();

        return(money);
    }
}
```

Typically, one lock object is associated with each resource that can potentially be used concurrently by multiple threads. Consequently, when a thread wants to operate on more than one resource at the same time, that thread will have to acquire the lock associated with each resource before proceeding. Any time

you have a design that requires a thread to grab more than one lock using separate lock-acquisition operations, you have the potential for deadlock. This can happen if more than one thread tries to acquire the same sets of locks in different order. Consider, for example, two threads T1 and T2 that are each going to grab locks L1 and L2 before operating on the two resources guarded by the two locks. Thread T1 is written to grab L1 then L2 before working on the resources. Thread T2 is written to acquire L2 first, followed by L1 before operating on the resources. Both of these approaches are equally valid when considered in isolation; the net result being that both locks are acquired before work is performed on the two resources being protected. However, when both threads are considered, the potential for deadlock exists. This is because (based on timing) it could happen that thread T1 grabs lock L1, then (before having a chance to grab lock L2), gets preempted by the scheduler. Now thread T2 could run and successfully grab lock L2. When thread T2 tries to subsequently grab lock L1 it will be blocked (because lock L1 is owned by thread T1). When thread T1 resumes and tries to grab lock L2, it will be blocked (because lock L2 is owned by thread T2). These two threads are now locked in a deadly embrace; each holding onto one lock while waiting for the other thread to release another lock. These two threads are deadlocked.

Most operating systems do not detect and actively break deadlock - doing so is extremely costly. Instead, programmers must prevent deadlock using either ordered lock acquisition (for example, making sure all threads grab lock L1 before L2 in the above example) or by using runtime-provided methods for grabbing multiple locks simultaneously. The `WaitHandle` class provides the `WaitAll` method for exactly this purpose. Figure 8.12 demonstrates how to safely (without fear of deadlock) grab ownership of two mutexes.

```
public class Bank {
    private Mutex accountLock = new Mutex();
    public void TransferFrom( Bank OtherBank, int HowMuch ) {
        Mutex[] accountLocks =
            {accountLock, OtherBank.accountLock};

        WaitHandle.WaitAll(accountLocks);

        Funds money = OtherBank.Withdraw(HowMuch);
        DepositToThisAccount(money);
        foreach( Mutex m in accountLocks )
            m.ReleaseMutex();
    }
    // Other methods not shown...
}

class App {
    static void Main() {
        Bank myAcct = new Bank();
        Bank yourAcct = new Bank();
        myAcct.TransferFrom(yourAcct, 1000);
    }
}
```

Figure 8.12: Using WaitHandle.WaitAll and mutexes

Events are typically used as condition variables that signal that "something has happened" - often the presence of work to be performed, or the notification of work completed. This means they are not used to provide serialized access to a shared resource in the same way a mutex or monitor would be used. However, they are useful for coordinating the activities of more than one thread at a much higher level. Like the mutex, events can be assigned a name and used from multiple processes.

Threading and COM Interop

Threads in the CLR can choose whether they want to live in the MTA or an STA

- Threads that never use COM interop don't really care
- Surprising thread switches across apartment boundaries can be devastating
- Performance hits
- Broken semantics
- Selection controlled via the `ApartmentState` property on the `Thread` class
- Can also apply `STAThread/MTAThread` attribute to `Main`
- Note: `WaitHandle.WaitAll` throws `NotSupportedException` on STA threads

Threads in the CLR need to be backed (at some point) by threads in the underlying operating system. Most of the time this fact isn't of interest to someone working purely in managed space, but when interoperating with COM components, the configuration of that underlying thread can be critical. In COM, each thread (that uses or is used by COM in some way) must select a home apartment. There are two types of apartments that threads can call home - single threaded apartments (STA) and the multithreaded apartment (MTA). If a thread chooses to live in an STA, it is declaring its intent to be the only thread to service all COM objects in that apartment. It also means that the thread may be used to service Windows messages by pumping a message queue. Both of these stipulations mean that an STA thread should not be blocked, unless there are no waiting messages for any COM objects or windows in that apartment. STAs are tricky beasts as a result. The MTA on the other hand has no such restrictions - many threads can live in the MTA and can service the COM objects there, and since no Windows will ever be created on an MTA thread (if you break this unspoken rule your life will be miserable), they can be blocked at any time, and can be created and destroyed at any time.

Objects in COM declare their apartment preference in the registry, via the `ThreadingModel` attribute. Suffice it to say that if a thread wants to talk to an object, and the object is in a different apartment from the calling thread, a thread switch generally must be made (there are exceptions to this rule, but this is often the case), which may wreak havoc in your system if it catches you off guard. Performance wise, thread switches are clearly to be avoided wherever possible. However, the semantics of your program might also become broken if you suffer a surprise thread switch: state stored in thread-local variables will be lost, and the security context of the thread will generally not be preserved (if you're impersonating Alice on the calling thread, after the thread switch, the new thread will most likely not be impersonating Alice).

One of the most important ways you can control the performance and semantics associated with apartment boundary thread switches is to simply take control over the type of apartment that the runtime should use for any given thread. Clearly this isn't important for threads that will never make calls through the interop layer. However, the first time a thread does dip down through COM interop, the runtime will need to select an apartment for that thread. This can only be done once (in COM, this is done by calling `CoInitializeEx` and selecting either an STA or MTA as home). The runtime simply consults a property (`ApartmentState`) of the thread to determine the type of apartment to choose. Figure 8.13 shows the various values this property can take.

```
public enum ApartmentState {  
    STA,  
    MTA,  
    Unknown    // default setting  
}
```

Figure 8.13: ApartmentState enumeration

In order to take control of the apartment type used by the thread, simply set this property before the thread makes any calls through the COM interop layer. If you're not sure if or when you'll call through COM interop for a given thread, set this property as early as possible (preferably right after the thread is created). Note that the runtime only consults this property when it needs to select a home apartment, and this only happens at most once per thread, so while you can change this property at any time, it won't have any affect if the thread has already made a call through COM interop. Just set this early and then don't mess with it; folks who maintain your code later will thank you for this. Note the default setting is Unknown - this simply is an indicator (to you) that you haven't yet selected your preference. If a thread is in this state when the runtime selects a home apartment for the thread, the runtime will select the MTA, so technically you only need to set this state explicitly if you want to force your thread into an STA. Figure 8.14 shows how this can be done for the main thread of an application. Figure 8.15 shows how this can be done using an attribute.

```
using System.Threading;  
  
class Foo  
{  
    static void Main()  
    {  
        Thread.CurrentThread.ApartmentState =  
            ApartmentState.STA;  
  
        // first call through COM interop will now cause  
        // the underlying OS thread to live in an STA.  
    }  
}
```

Figure 8.14: Selecting an apartment type for the main thread


```
using System;  
  
class Foo  
{  
    [STAThread]  
    static void Main()  
    {  
    }  
}
```

Figure 8.15: Selecting an apartment type for the main thread

Summary

- Async method invocation using delegates are the preferred threading mechanism
- The ThreadPool and Timer classes can be used for general purpose callback requirements
- Threads can be created and managed explicitly by the programmer
- Several low-level primitives are provided for ensuring proper thread synchronization
- The Monitor and ReaderWriterLock provide higher order synchronization support
- Threads that use COM interop should select the most appropriate apartment type

Module 9

AppDomains and Marshaling

App Domains act as mini-processes within the CLR. AppDomains provide an execution boundary for fault isolation, type identity, security policy, and more. AppDomains also act as the remoting and marshaling boundary between components and web services.

After completing this module, you should be able to:

- ❑ understand how types and objects are scoped
- ❑ create and manage multiple AppDomains
- ❑ expose CLR-based objects as web services

AppDomains

App Domains act as mini-processes in the CLR. They provide an execution boundary for fault isolation, types, security identity, and more.

Execution scope and the CLR

Running applications are modeled in the CLR as AppDomains

- AppDomains fill the role of the OS process in the CLR
- An AppDomain is scoped to a particular process/runtime
- A process/runtime can host multiple AppDomains
- AppDomains are cheaper than OS processes
- An OS thread can switch AppDomains much faster than a process switch
- An object's type controls cross-appdomain marshaling

AppDomains fill many of the same roles filled by an operating system process. AppDomains, like processes, scope the execution of code. AppDomains, like processes, provide a degree of fault isolation. AppDomains, like processes, provide a degree of security isolation. AppDomains, like processes, own resources on behalf of the programs they execute. In general, most of what you may know about an operating system process probably applies to AppDomains.

AppDomains are strikingly similar to processes, yet they are ultimately two different things. A process is an abstraction created by your operating system. An AppDomain is an abstraction created by the CLR. While a given AppDomain resides in exactly one OS process, a given OS process can host multiple AppDomains. This relationship is shown in figure 9.1.

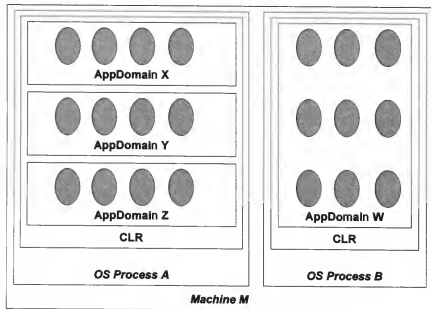


Figure 9.1: Objects, AppDomains, and Processes

To reinforce the isolation between AppDomains, each AppDomain runs with its own "top-of-stack" permission set, which allows different domains in a single process to run with different security permissions. This feature is critical in ISP/hosting environments where applications from multiple users need to run simultaneously but the expense of a process-per-application is not feasible.

It is less costly to create an AppDomain than it is to create an OS process. Similarly, it is cheaper to cross AppDomains boundaries than it is to cross OS process boundaries. However, as is the case with OS processes, it is difficult

(but not impossible) to share data between AppDomains. One reason that sharing is difficult is due to the way that objects and AppDomains relate.

An object resides in exactly one AppDomain, as do values. Moreover, object references must refer to objects in the same AppDomain. In this respect, AppDomains behave as if they have their own private address space. However, this behavior is only an illusion, as all it takes is an unmanaged or unverifiable method to start walking over memory to shatter this illusion. If only managed, verifiable code is executed, then this illusion is in fact the rule of the land.

Like objects, types reside in exactly one AppDomain. If two AppDomains need to use a type, the type must be initialized and allocated once per AppDomain. Additionally, the type's module and assembly must also be loaded and initialized once for each AppDomain the type is used in. Because each AppDomain in a process maintains a separate copy of the type, each AppDomain has its own private copy of the type's static fields. Figure 9.2 shows the relationship between AppDomains, objects, and types.

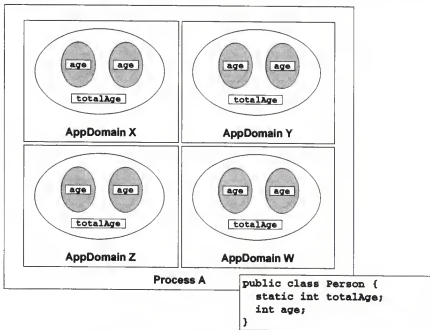


Figure 9.2: Scoping types and statics to AppDomains

Like processes, AppDomains are a unit of ownership. The resources that are owned by an AppDomain include loaded modules, assemblies, and types. These resources are held in memory as long as the owning AppDomain is loaded. Unloading an AppDomain is the only way to unload a module or assembly.

Unloading an AppDomain is also the only way to reclaim the memory consumed by a type's static fields.

AppDomains provide protection for code. When an exception is thrown in one AppDomain that is not handled by application code, an AppDomain-specific unhandled exception handler can dispatch the exception without terminating the process. This allows other domains to remain running despite the fact that one domain in the process has failed.

It is difficult to talk about processes without quickly steering the conversation to the topic of threads. The CLR has its own abstraction for modeling the execution of code that is conceptually similar to an OS thread. The CLR defines a type, `System.Threading.Thread` that represents a schedulable entity in an AppDomain. A `System.Threading.Thread` thread object is sometimes referred to as a "soft thread," as it is a construct that is not recognized by the underlying operating system. In contrast, OS threads are referred to as "hard threads" as they are what the OS deals with.

There is not a one-to-one relationship between hard threads and CLR soft thread objects. However, certain realities are known to be true based both on the programming model as well as empirical analysis of the current CLR implementation. For one, a CLR soft thread object resides in exactly one AppDomain. This is a byproduct of how AppDomains work and what they mean and must be true no matter how the CLR's implementation changes over time. Secondly, a given AppDomain may have multiple soft thread objects. In the current implementation, this happens when two or more hard threads execute code in a single AppDomain. All other assumptions about the soft/hard thread relationship are implementation specific. With that disclaimer in place, there are a few other observations worth noting.

In the current implementation of the CLR, a given hard thread will have at most one soft thread object affiliated with it for a given AppDomain. Also, if a hard thread winds up executing code in multiple AppDomains, each AppDomain will have a distinct soft thread object affiliated with that thread. However, if a hard thread never enters a given AppDomain, then that AppDomain will not have a soft thread object that represents the hard thread. These observations are illustrated in figure 9.3.

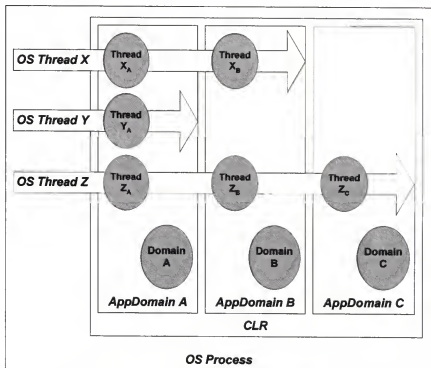


Figure 9.3: AppDomains and threads

Finally, each time a given hard thread enters an AppDomain, it gets the same soft thread object. Again, please note that these observations are based on the behavior of the current CLR implementation. In particular, when the CLR is hosted in a fiber-based environment (such as SQL Server), it is likely that one or more of these assumptions may break, most likely the assumption that a hard thread has at most one soft thread object per AppDomain.

The CLR maintains a fair amount of information in the hard thread's thread-local-storage (TLS). In particular, references to the current AppDomain and soft thread object can be found in hard TLS. When a hard thread crosses over from one AppDomain to another, the CLR automatically adjusts these references to point to the new "current" AppDomain and soft thread. The current implementation of the CLR maintains a per-AppDomain thread table to ensure that a given hard thread is only affiliated with one soft thread object per AppDomain.

It is worth noting that a soft thread object has its own private thread-local-storage that is accessible via the `Thread.GetData/Thread.SetData` methods. Because this TLS is bound to the soft thread object, when a hard

thread crosses AppDomain boundaries, it cannot see the soft TLS stored while executing in the previous AppDomain.

Programming with AppDomains

AppDomains can be programmed explicitly via the `System.AppDomain` type

- `AppDomain.CurrentDomain` static property accesses current domain
- `CreateDomain` creates a new AppDomain in the current OS process
- `Unload` stops an AppDomain and releases its resources
- `ExecuteAssembly` runs a program in the target domain
- `DoCallback` invokes a delegate in the target AppDomain
- `GetData/SetData` act as domain-specific environment variables

AppDomains are exposed to programmers via the `System.AppDomain` type. A subset of the public signature of `System.AppDomain` is shown in figure 9.4. The most important member of this type is the `CurrentDomain` static property. This property simply fetches the AppDomain reference that is stored in the hard thread's TLS. As a point of interest, the "current" soft thread object can be extracted from the hard thread's TLS via the `Thread.CurrentThread` property.

```
public sealed class AppDomain {
    public static AppDomain CurrentDomain { get; };

    public object GetData(string name);
    public void SetData(string name, object value);

    public Assembly[] GetAssemblies();

    // Note: CreateDomain has several overloads
    public static AppDomain CreateDomain(
        string friendlyName,      // name for debugger
        Evidence secinfo,         // top-of-stack sec
        AppDomainSetup ldrinfo);  // fusion env vars
    public static void Unload(AppDomain target);

    // Note: ExecuteAssembly has several overloads
    public void DoCallback(CrossAppDomainDelegate cb);
    public int ExecuteAssembly(string assemblyFile,
                               Evidence assemblySecurity,
                               string[] argv);
    public object CreateInstanceAndUnwrap( string asmName,
                                           string typeName );
    // Other members not shown...
}
```

Figure 9.4: `System.AppDomain` (excerpt)

Once you have a reference to an AppDomain, there are a variety of things you can do with it. For one, each AppDomain has its own set of environmental properties that are accessible via the `SetData` and `GetData` methods. These properties act like the environment variables of an OS process, but unlike process environment variables, these properties are scoped to a particular AppDomain. These properties are functionally equivalent to static fields; however, unlike static fields, they are not duplicated per assembly or assembly version, which makes them a handy replacement for static fields in side-by-side versioning scenarios.

AppDomains can be created and destroyed programmatically. While this is normally done by hosting environments like ASP.NET, your application can access these same facilities to spawn new AppDomains. The

.exe == .dll except .exe has an entry point

`AppDomain.CreateDomain` method creates a new `AppDomain` in the current process and returns a reference to the new domain. The domain will remain in memory until a call to `AppDomain.Unload` causes it to be removed from memory. Once an `AppDomain` has been created, you can force it to load and execute code using a variety of techniques. The most direct way to do this is using the `AppDomain.ExecuteAssembly` method.

`AppDomain.ExecuteAssembly` causes the target `AppDomain` to load an assembly and execute its main entry point. The specified assembly will not be loaded or initialized in the parent `AppDomain`; rather, `ExecuteAssembly` will first switch `AppDomains` to the child domain and load and execute the code while running in the child domain. If the specified assembly calls `AppDomain.CurrentDomain`, it will get the child `AppDomain` object. If the specified assembly uses any of the same assemblies as the parent program, the child `AppDomain` will load its own independent copies of the types and modules, including its own set of static fields.

Figure 9.5 shows an example of the `ExecuteAssembly` method in action. `ExecuteAssembly` is a synchronous routine. That means that the caller is blocked until the child program's `Main` method returns control to the runtime. One can use the asynchronous method invocation mechanism discussed in a previous chapter if non-blocking execution is desired.

using System;

```
public class MyApp {  
    public static int Main(string[] argv) {  
        // create domain  
        AppDomain child = AppDomain.CreateDomain("childapp");  
        // execute yourapp.exe  
        int r = child.ExecuteAssembly("yourapp.exe", null, argv);  
        // unload domain  
        AppDomain.Unload(child);  
        // return result  
        return r;  
    }  
}
```

Figure 9.5: Spawning new applications

It is also possible to inject arbitrary code into an `AppDomain`. The `AppDomain.DoCallback` method allows you to specify a method on a type that will be executed in the foreign domain. The specified method should be static and have a signature that matches the `CrossAppDomainDelegate`'s signature. Additionally, the specified method's type, module and assembly will need to be loaded in the foreign `AppDomain` in order to execute the code. If the specified method needs to share information between the `AppDomains`, it

can use the `SetData/GetData` methods on the foreign `AppDomain`. Figure 9.6 shows an example of this technique.

```
using System;
using System.Reflection;

public class MyApp {
    static void Main() {
        AppDomain child = AppDomain.CreateDomain("childapp");
        Console.WriteLine(GetAssemblyCount(child));
        AppDomain.Unload(child);
    }
    static int GetAssemblyCount(AppDomain target) {
        // create the delegate
        CrossAppDomainDelegate cb =
            new CrossAppDomainDelegate(CallbackProc);
        // inject and execute the code
        target.DoCallBack(cb);
        // extract the property
        return (int)target.GetData("LoadedAsmCount");
    }
    public static void CallbackProc() {
        AppDomain current = AppDomain.CurrentDomain;
        current.SetData("LoadedAsmCount",
            current.GetAssemblies().Length);
    }
}
```

Figure 9.6: Calling into foreign AppDomains

AppDomain events

An AppDomain can raise events at well-known times

- Some events are related to the assembly loader
- Other events are related to termination
- Can be notified of successful assembly loads
- Can act as "last-chance" assembly resolver when code cannot be found

The `AppDomain` type supports a handful of events that allow interested parties to be notified of significant conditions in a running program. These events are listed in figure 9.7. Four of these events are related to the assembly resolver and loader. Three of these events are related to terminal conditions in the process. `UnhandledException` has already been discussed in the *Method Invocation I* chapter. `DomainUnload` is called just prior to the unloading of an `AppDomain`. `ProcessExit` is called just prior to the termination of the CLR in a process.

EventName	EventArgs Properties	Description
AssemblyLoad	Assembly LoadedAssembly	Assembly had just been successfully loaded
AssemblyResolve	string Name	Assembly reference cannot be resolved
TypeResolve	string Name	Type reference cannot be resolved
ResourceResolve	string Name	Resource reference cannot be resolved
DomainUnload	None	Domain is about to be unloaded
ProcessExit	None	Process is about to shutdown
UnhandledException	bool IsTerminating, object ExceptionObject	Exception escaped thread-specific handlers

Figure 9.7: AppDomain events

There are four `AppDomain` events related to assembly resolution and loading. One of the events (`AssemblyLoad`) is used to notify interested parties when a new assembly has successfully been loaded. The other three of these events are used by the assembly resolver when a type (`TypeResolve`), assembly (`AssemblyResolve`), or manifest resource (`ResourceResolve`) cannot be resolved. For these three events, the event handler is given the opportunity to produce a `System.Reflection.Assembly` object that can be used to satisfy the request. It is important to note that these three methods are only called after the resolver has gone through its standard techniques for finding the desired assembly. These events are primarily useful for implementing an application-specific "last-chance" assembly resolver that uses some app-specific policy for converting the requested `AssemblyName` into a codebase that can be passed to `Assembly.LoadFrom`.

Figure 9.8 shows an example that uses the `AssemblyResolve` event to supply a backup policy for finding assemblies. In this example, the simple name of the requested assembly is munged into an absolute pathname into the `windows\system32` directory. Once the new pathname is constructed, the event handler passes control to the low-level loader using the `Assembly.LoadFrom` method.

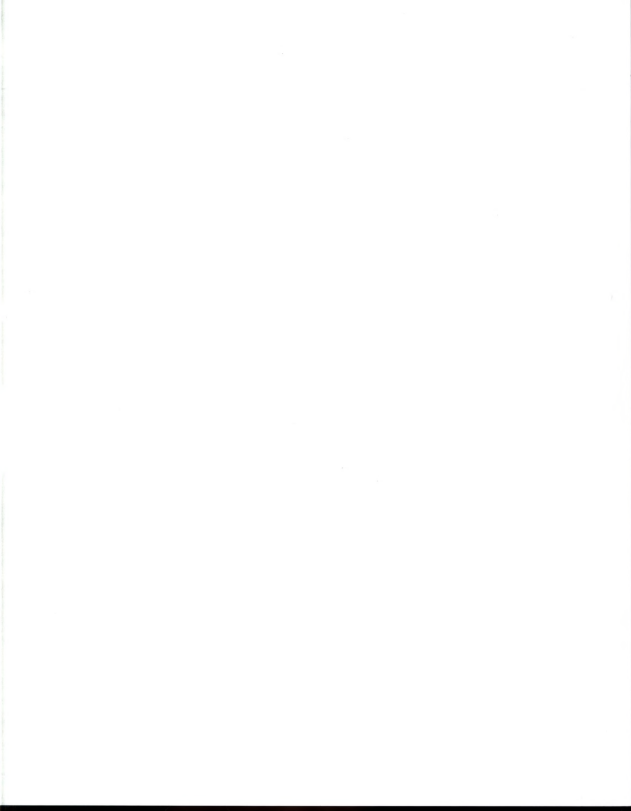
```
using System;
using System.Reflection;

class oldhack {
    static oldhack() {
        // register the event handler at type-init time
        AppDomain.CurrentDomain.AssemblyResolve +=
            new ResolveEventHandler(Backstop);
    }
    static Assembly Backstop(object sender,
                             ResolveEventArgs args) {
        // extract the simple name from the display name
        string displayName = args.Name;
        string simpleName = displayName.Split(',')[0];

        // build the retro-file name
        string fullPath = String.Format(
            "C:\\windows\\system32\\{0}.dll", simpleName);

        // delegate to LoadFrom
        return Assembly.LoadFrom(fullPath);
    }
    static void Main() {
        Console.WriteLine(MyCode.GetIt());
    }
}
```

Figure 9.8: Retro-programming in C#



AppDomains and the assembly resolver

Assembly resolver makes extensive use of AppDomain properties

- Can configure most resolver properties for each AppDomain
- Can configure AppDomain to use a dynamic base directory for codegen scenarios
- Can configure AppDomain to use a shadow copying to avoid file locking

AppDomains play a critical role in controlling the behavior of the assembly resolver. Most of the assembly resolver's behavior is controlled by the AppDomain's properties. In fact, the properties used by the assembly resolver are all stored in a single data structure of type `AppDomainSetup`, which is maintained on a per-AppDomain basis. This data structure is exposed via the `AppDomain.SetupInformation` property.

Each AppDomain can have its own APPBASE and configuration file. By virtue of this fact, each AppDomain can have its own probe path and version policy. The properties used by the assembly resolver can be set either using the AppDomain's `AppDomainSetup` property or by calling `SetData/GetData` with the right well-known property name.

The properties of an AppDomain that are used by the assembly resolver are shown in figure 9.9. This table shows both the `AppDomainSetup` member name as well as the well-known name to use with `AppDomain.SetData`. Several of the properties shown in this table have already been described in the modules and assemblies chapter. However, there are two sets of properties that have not yet been discussed. One set alters the probe path; the other set controls how code is actually loaded.

AppDomainSetup Property	Get/SetData Name	Description
ApplicationBase	APPBASE	Base directory for probing
ApplicationName	APP_NAME	Symbolic name of application
ConfigurationFile	APP_CONFIG_FILE	Name of .config file
DynamicBase	DYNAMIC_BASE	Root of codegen directory
PrivateBinPath	PRIVATE_BINPATH	Semi-colon-delim list of subdirs
PrivateBinPathProbe	BINPATH_PROBE_ONLY	Suppress probing at APPBASE ("*" or null)
ShadowCopyFiles	FORCE_CACHE_INSTALL	Enable/disable shadow-copy (boolean)
ShadowCopyDirectories	SHADOW_COPY_DIRS	Directories to shadow-copy from
CachePath	CACHE_BASE	Directory to shadow-copy to
LoaderOptimization	LOADER_OPTIMIZATION	JIT-compile per-process or per-domain

AppDomain Property	Description
BaseDirectory	Alias to <code>AppDomainSetup.ApplicationBase</code>
RelativeSearchPath	Alias to <code>AppDomainSetup.PrivateBinPath</code>
DynamicDirectory	Directory for dynamic assemblies (<DynamicBase>\<ApplicationName>)
FriendlyName	Name of AppDomain used in debugger

Figure 9.9: AppDomain environment properties

Recall that when an assembly is not found in the GAC or via a codebase hint, the assembly resolver looks in the probe path of the application. This path was set using the probing element in the configuration file, and is visible programmatically via the `AppDomain.RelativeSearchPath`. Also recall that this relative search path is in fact relative; it cannot refer to directories that are not children of the `APPBASE` directory.

Now consider the case where an application needs to generate code dynamically. If that code is to be stored on disk, then it is an open question as to where it should be stored. If the generated assembly is to be loaded via probing (which is likely if the code is specific to the generating application), then the application must have write access to a directory underneath `APPBASE`. However, there are many scenarios in which it is desirable to execute an application from a read-only part of a file system (e.g., a secured server or CD-ROM), which means that some alternative location needs to be used for dynamically generated code. This is the role of the `AppDomain.DynamicDirectory` property.

Each `AppDomain` can have at most one dynamic directory. The dynamic directory is searched during probing prior to looking in the probe path specified by the probing element in the configuration file. The dynamic directory is derived by concatenating two other properties of the `AppDomain`: `APP_NAME` (a.k.a. `AppDomainSetup.ApplicationName`) and `DYNAMIC_BASE` (a.k.a. `AppDomainSetup.DynamicBase`). This feature is used extensively by ASP.NET, which is one of the more notorious generators of code. On the author's machine, the `DYNAMIC_BASE` property for my default virtual directory is currently:

```
C:\Program Files\ASP.NET\Premium\V1.0Beta2\Temporary
ASP.NET Files\root\fa7064c6
```

The `APP_NAME` for the web application running in my default virtual directory is currently:

```
2014c0f1
```

This means that the resultant `DynamicDirectory` for that web application is:

```
C:\Program Files\ASP.NET\Premium\V1.0Beta2\Temporary
ASP.NET Files\root\fa7064c6\2014c0f1
```

Every DLL that the ASP.NET engine generates for that web application is stored in this directory. Because this directory is searched as part of the probing process, DLLs found in that directory can be successfully loaded, despite the fact that they are not under the `APPBASE` for the `AppDomain` (which in this case is `C:\inetpub\wwwroot`). As a point of interest, ASP.NET sets the `BINPATH_PROBE_ONLY` property to suppress probing in the `APPBASE` directory. This is why you cannot simply put a DLL into your virtual directory and get ASP.NET to find it. Rather, ASP.NET presets the probe path to `bin`,

which is where any pre-built DLLs used by an ASP.NET application must be stored.

The second set of AppDomain properties that warrant discussion relate to a feature known as shadow copying. Shadow copy addresses a common (and annoying) problem related to server-side development and deployment. Prior to .NET, developing and deploying DLLs that load into server-side container environments (e.g., IIS, COM+) has been somewhat problematic due to the way the classic Win32 loader worked. When the Win32 loader loads a DLL, it takes a read lock on the file to ensure that no changes are made to the underlying executable image. Unfortunately, this means that once a DLL is loaded into a server-side container, there is often no way to overwrite the DLL with a new version without first shutting down the container to release the file lock. This problem is solved by shadow copying.

When an assembly is loaded using shadow copying, a temporary copy of the underlying files is made in a scratch directory and the temporary copies are loaded in lieu of the "real" assembly files. When shadow copying is enabled for an AppDomain, you must specify a directory path: `SHADOW_COPY_DIRS`. `SHADOW_COPY_DIRS` (a.k.a. `AppDomainSetup.ShadowCopyDirectories`) indicates the parent directories of the assemblies that you want to be shadow copied.

Figure 9.10 shows a simple application that displays the location from which it's assembly was loaded. When this application is loaded into an appdomain that does not have shadow copying enabled, you will not be able to delete or replace the assembly file because it is opened in-situ while the application is running. If, on the other hand, the application is loaded into an appdomain that has shadow copying enabled, the application assembly file will be copied by the runtime to a scratch directory location and opened from that location instead - allowing the original version of the assembly file to be deleted or replaced with a newer version. Figure 9.11 shows a simple host application that creates an appdomain with shadow copying enabled and then executes the child application within that appdomain. Figure 9.12 illustrates what happens as result of running the host application shown previously.

```
using System;
using System.Reflection;

class App {
    static void Main() {
        AppDomain currentDomain = AppDomain.CurrentDomain;
        Console.WriteLine("My domain: {0}",
            currentDomain.FriendlyName);
        Console.WriteLine("Assembly location:");

        Console.WriteLine(Assembly.GetExecutingAssembly().Location)
        ;
        Console.WriteLine("Try to delete me now");
        Console.ReadLine();
    }
}
```

Figure 9.10: A simple application (child.exe)

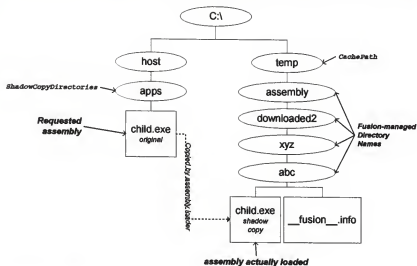
```
using System;

class Host {
    static void Main() {
        Console.WriteLine( "Host app running in appdomain
{0}",
AppDomain.CurrentDomain.FriendlyName );

        // Shadow copy any assemblies loaded from the 'apps'
        // subdirectory of this application's APPBASE.
        AppDomainSetup adSetup = new AppDomainSetup();
        adSetup.ShadowCopyFiles = "true";
        adSetup.ShadowCopyDirectories =
            AppDomain.CurrentDomain.BaseDirectory + "apps\\";

        AppDomain ad = AppDomain.CreateDomain("childDomain",
null,
adSetup);
        ad.ExecuteAssembly("apps\\child.exe");
    }
}
```

Figure 9.11: Implementing shadow copy (host.exe)



Assembly.CodeBase	file:///C:/host/apps/child.exe
Assembly.Location	C:/temp/domainFromHell/assembly/downloaded2/xyz/abc/child.exe

Figure 9.12: Loading using shadow copy

Once again, ASP.NET is a heavy user of this feature, which is appropriate given its status as the de facto server-side container for the CLR. On the author's machine, the `SHADOW_COPY_DIRS` property simply points to the web application's bin directory (C:\inetpub\wwwroot\bin to be exact). The `CACHE_BASE` points to the same directory as the `DYNAMIC_BASE` property.

When shadow copying is in use, the assembly's `CodeBase` will still match the original location of the assembly manifest. This is important for code-access-security, which is discussed in the next chapter. To discover the actual path used to load the assembly, one can use the `Assembly.Location` property, as is shown in figure 9.12.

AppDomains and Objects Revisited

Passing object references across AppDomain boundaries requires marshaling

- Types that derive from `System.MarshalByRefObject` produce AppDomain-bound objects
- Types marked `[Serializable]` are unbound and marshal by value
- Happens transparently when calling certain AppDomain properties/methods
- Happens transparently when passing an object as parameter during remoting
- No other types may be marshaled

This chapter began by framing AppDomains as scopes of execution. A large part of that discussion was dedicated to portraying an AppDomain as a "home" for objects and types. In particular, an object is scoped to a particular AppDomain and object references may only refer to objects in the same AppDomain. However, there is a slight inconsistency in the AppDomain interface that has yet to be discussed. That slight inconsistency is the `SetData/GetData` mechanism.

Figure 9.6 showed an example of injecting code into a foreign AppDomain. In that example, the `SetData/GetData` mechanism was used to pass the number of loaded assemblies from one AppDomain to another. In reviewing the signatures of these two methods:

```
static public void SetData(string name, object value);
static public object GetData(string name);
```

it would appear that an object reference can be stored into a common property from one AppDomain and fetched (and used!) from another. In fact, that is exactly what the code in Figure 9.6 did. One might ask how an object reference from one AppDomain can be smuggled into another domain given that object references are AppDomain-specific. The answer is marshaling.

Objects, values, and object references are all scoped to a particular AppDomain. When a reference or value needs to be passed to another AppDomain, it must first be marshaled. Much of the CLR's marshaling infrastructure is in the `System.Runtime.Remoting` namespace. In particular, the type `System.Runtime.Remoting.RemotingServices` has two static methods that are fundamental to marshaling: `Marshal` and `Unmarshal`.

When marshaling an object reference, the concrete type of the object determines how marshaling will actually work. As shown in figure 9.13, there are three possible scenarios.

Category	AppDomain-bound	Unbound	Remote-unaware
Applicable Types	Types derived from <code>MarshalByRefObject</code>	Types marked <code>[Serializable]</code>	All other types
Cross-domain marshaling behavior	Marshal-by-reference across AppDomain	Marshal-by-value across AppDomain	Cannot leave AppDomain
Inlining Behavior	Inlining disabled to support proxy access	Inlining enabled	Inlining enabled
Proxy Behavior	Cross-domain proxy Same-domain direct	Never has a proxy	Never has a proxy
Cross-domain identity	Has distributed identity	No distributed identity	No distributed identity

Figure 9.13: Agility and objects

By default, types are remote-unaware and do not support cross-AppDomain marshaling. Attempts to marshal instances of a remote-unaware type will fail.

If a type derives from `System.MarshalByRefObject` either directly or indirectly, then it is AppDomain-bound. Instances of AppDomain-bound types will marshal by reference. This means that the receiver of the marshaled object (reference) will be given a proxy that remotes (forwards) all member access back to the object's home AppDomain. Technically, only access to instance members will be remoted back to the object's home AppDomain. Static methods are never remoted.

Types that do not derive from `MarshalByRefObject` but do support object serialization (indicated via the `[System.Serializable]` pseudo-custom attribute) are considered unbound to any AppDomain. Instances of unbound types will marshal by value. This means the receiver of the marshaled object (reference) will be given a disconnected clone of the original object. All three behaviors are illustrated in figure 9.14.

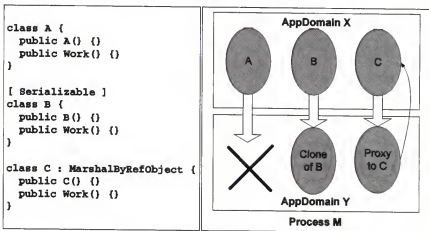


Figure 9.14: Marshaling objects across AppDomains

Consider the sample code shown in figure 9.15, which creates a new AppDomain and then proceeds to instantiate each of the 3 target types described in figure 9.14 within that appdomain, fetching an object reference in return, and ask yourself two questions for each type being demonstrated: (1) in which AppDomain will the constructor for the type execute? and (2) in which AppDomain will the subsequent methods on the new object execute?

```
using System;

class App {
    static void Main() {
        // Ask yourself these questions for each type below:
        // (1) In which appdomain will the constructor run?
        // (2) In which appdomain will the Work method run?
        //
        AppDomain sandbox = AppDomain.CreateDomain("sandbox");

        A a = (A)sandbox.CreateInstanceAndUnwrap("abclib",
"A");
        a.Work();

        B b = (B)sandbox.CreateInstanceAndUnwrap("abclib", "B")
b.Work();

        C c = (C)sandbox.CreateInstanceAndUnwrap("abclib",
"C");
        c.Work();
    }
}
```

Figure 9.15: Cross-AppDomain marshaling

Marshaling typically happens implicitly when a call is made to a cross-AppDomain proxy. The input parameters to the method call are marshaled into a serialized request message that is sent to the target AppDomain. When the target AppDomain receives the serialized request, it first deserializes the message and pushes the parameters onto a new stack frame. After the method has been dispatched to the target object, the output parameters and return value are then marshaled into a serialized response message that is sent back to the caller's AppDomain where they are unmarshaled and placed back on the caller's stack. These mechanics are handled transparently by the .NET Remoting infrastructure of the CLR.

Summary

- AppDomains scope types and objects at runtime
- AppDomains isolate resource reclamation and unhandled exceptions
- AppDomains heavily influence the assembly resolver
- Some objects can be marshaled across AppDomain boundaries

Module 10

Serialization

After completing this module, you should be able to:

- use formatters to serialize object graphs into binary or XML
- leverage metadata-driven serialization
- deal with transient data
- implement custom serialization

Serialization

Serialization is the act of traversing a connected graph of objects, converting each object's state into a flat, context-neutral format. Deserialization is the procedure performed in reverse - reading the stored state and reconstituting a connected graph of objects in memory.

Motivation

Why serialize?

- Persist object state for later retrieval
- Pass objref by value across remoting boundaries
- Opportunity to control bandwidth/space-requirements of such operations

Serialization is generally useful in two scenarios. The first, typically referred to as "persistence", is the act of saving or storing an object's state (or a graph of objects) to/from a persistent medium of some kind (like a file or database). Serialization is also important in remoting in terms of defining exactly what happens when a reference to an object is passed across a remoting boundary.

Serialization defined

The conversion of an object graph to/from a context-neutral representation

- Object graph is a set of connected object references
- Serialization involves visiting each object in the graph, writing out type information + state.
- Deserialization involves reading stored type information, instantiating that type, initializing object from stored state, connecting object to graph being built.

When, what, how?

There are three aspects to serialization

- When: client-driven persistence or objref marshaling between AppDomains
- What: only types marked [Serializable]
- How: formatters control layout of serialized data in stream

ctrl-space - to complete anything -

Serialization is a client-driven activity. That client might be a programmer that wants to save or restore an object's state to a file, or the runtime attempting to pass an object reference across a remoting boundary.

Only types that advertise themselves as serializable can be serialized. There are two ways to indicate your type is serializable. The simplest is to apply the `[Serializable]` attribute, as shown in figure 10.1. In this case, the metadata describing your type drives the serialization process. During serialization, reflection will be used to retrieving the type description for your object, along with the contents of each field of your type. During deserialization, reflection will again be used to activate an instance of your type and then set the contents of each field to their previously stored state.

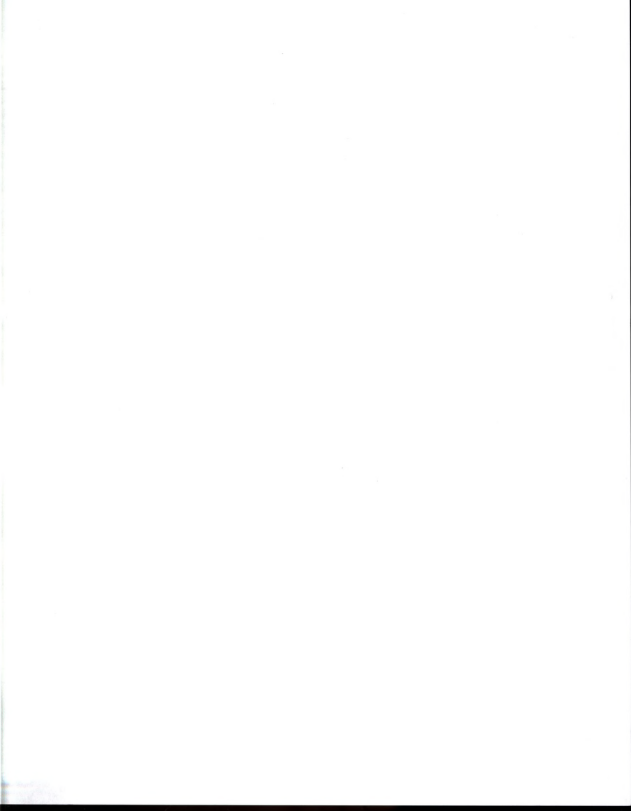
```
[Serializable]
public class DigitSeries {
    public DigitSeries( int first, int last ) {
        start = first;
        end = last;
        GenerateSeries();
    }
    private void GenerateSeries() {
        int count = end - start + 1;
        digits = new int[count];
        for( int n = 0; n < count; n++ )
            digits[n] = start + n;
    }
    private int    start; // Starting value.
    private int    end;   // Ending value.
    private int[]  digits; // The actual digits.
}
```

Figure 10.1: Case study

The second way to indicate that your type is serializable is to implement the `ISerializable` interface (in addition to marking your class `[Serializable]`). This is referred to as custom serialization, because you have total control over what information get serialized, or how that stored information gets deserialized. Custom serialization will be discussed a little later in this module.

During serialization, the information being serialized is always written to an instance of a `System.Stream` derived class. Although the object controls what is serialized, a formatter always controls the layout of the serialized information within the stream. Formatters implement the `IFormatter` interface, or extend the `Formatter` abstract class. The runtime includes two standard formatters: the `SoapFormatter` class and the `BinaryFormatter` class. Custom formatters can also be built by implementing the `IFormatter`

interface. You may also extend the `Formatter` abstract class to leverage some common implementation.



Client-driven serialization

The serialization process is the same whether or not the client is a programmer on your team or a part of the runtime's remoting infrastructure.

- Client creates/opens Stream of some kind
- Client instantiates Formatter
- Client tells formatter to serialize object to the stream
- Deserialization similar, with formatter returning reconstituted object graph

During serialization, a client creates or opens a stream and instantiates some kind of formatter. Then the client asks the formatter to serialize a given object to that stream. Figure 10.2 illustrates this using the `SoapFormatter` class.

```
using System.IO;
using System.Runtime.Serialization;
using System.Runtime.Serialization.Formatters.Soap;

void SaveSeriesToFile( String file, DigitSeries series )
{
    FileStream fs = new FileStream(file, FileMode.Create);
    SoapFormatter sf = new SoapFormatter();
    sf.Serialize(fs, series);
}
```

Figure 10.2: Client-driven serialization

During deserialization, a client instantiates a formatter and opens a stream that was previously used to serialize an object graph. Then the client asks the formatter to deserialize the object graph contained in that stream. If successful, the formatter returns a reference to the completely reconstituted object graph in memory. Since the formatter always returns the object reference as a `System.Object` return value, you will always cast that return value to the type of the object being deserialized. Figure 10.3 illustrates this.

```
using System.IO;
using System.Runtime.Serialization;
using System.Runtime.Serialization.Formatters.Soap;

DigitSeries ReadSeriesFromFile( String file )
{
    FileStream fs = new FileStream(file, FileMode.Open);
    SoapFormatter sf = new SoapFormatter();
    return (DigitSeries)sf.Deserialize(fs);
}
```

Figure 10.3: Client-driven deserialization

Transient data

Sometimes it doesn't make sense to store everything

- Password buffer, read-ahead cache
- Some state might be dynamically generated
- Trade off space for time during remoting
- [NonSerialized] attribute denotes throw-away field
- IDeserializationCallback allows fixups to transient fields when deserialized

Consider our `DigitSeries` example class. When an instance of this object is passed by value across a remoting boundary (which may be over a low-bandwidth connection), the amount of information transmitted could be potentially very large (the first number in the series, the last number in the series, and every number in between). For this particular class, we call tell the formatter to skip the integer array member `digits` by applying the `[System.NonSerialized]` attribute. This will conserve bandwidth or file/database space during serialization. Any field that is marked as `[NonSerialized]` will be skipped during serialization and will be set to zero (for scalars) or null (for references) upon deserialization.

If a `[NonSerialized]` member needs to be fixed up after deserialization occurs, but before the reconstituted object is returned by the formatter for subsequent usage, you can implement the `IDeserializationCallback` interface. This interface has only one method (`OnDeserialization`), which will be called after the formatter has reconstructed the object, but before object is returned from the `Deserialize` method. This is your opportunity to perform any fixups that may be required. Figure 10.4 shows how a revised `DigitSeries` class that uses the `[NonSerialized]` attribute on the `digits` array and then implements the `IDeserializationCallback` interface to fixup that array when deserialization occurs.

```
using System;
using System.Runtime.Serialization;

[Serializable]
public class DigitSeries : IDeserializationCallback
{
    // ...
    [NonSerialized] private int[] digits;

    public virtual void OnDeserialization( Object sender )
    {
        GenerateSeries();
    }
}
```

Figure 10.4: `[NonSerialized]` and `IDeserializationCallback`

Custom serialization

Sometimes metadata-driven serialization doesn't meet your serialization needs

- The majority of your fields are [NonSerialized] and lots of fixups being performed in OnDeserialization
- Object anatomy might change between serialization and deserialization
- Implement ISerializable interface in addition to applying [Serializable] attribute to perform custom serialization
- Interface has one method (GetObjectData) and implies a special "deserialization constructor"

When it's possible that the anatomy of your type might change between the time when your type is serialized, and later deserialized, then metadata-driven serialization isn't sufficient. In this case, you'll need to store some kind of schema number along with the type information and data information. During deserialization, you can then consult the schema number to determine what "shape" your object had at the time it was serialized, and react accordingly.

Figure 10.5 shows a revised `DigitSeries` class that has implemented the `ISerializable` interface. The `GetObjectData` method is called during serialization. In this example, we simply add each of our member fields (including the array) to the `SerializationInfo` collection. The `SerializationInfo` collection provides a collection that associates named keys with associated values. In this example, the named keys we use are the same as the field names for our class - but this is not required. This example also shows the format of the deserialization constructor. This constructor, which is used to instantiate your object during deserialization, is presented with a `SerializationInfo` collection. You simply retrieve what you put into the collection by specifying the named keys used by `GetObjectData`. For standard types, there are formatted helpers (like `GetInt32`) that return the value pre-cast to the appropriate type. For more complex types, like our array of integers, you use the generic `GetValue` function to fetch values. This function returns an object, which you then cast to the appropriate type.

```
[Serializable]
public class DigitSeries : ISerializable {
    // ...
    public DigitSeries( SerializationInfo si,
                       StreamingContext sc )
    {
        start = si.GetInt32("start");
        end = si.GetInt32("end");
        digits =
            (int[])si.GetValue("digits", typeof(int[]));
    }
    public void GetObjectData( SerializationInfo si,
                              StreamingContext sc )
    {
        si.AddValue("start", start);
        si.AddValue("end", end);
        si.AddValue("digits", digits);
    }
}
```

Figure 10.5: Implementing `ISerializable`

Summary

- Types marked [Serializable] support metadata-driven serialization
- Fields marked [NonSerialized] are skipped during metadata-driven serialization
- Implement IDeserializationCallback to fixup transient fields
- Implement ISerializable to perform custom serialization



